HELSINKI UNIVERSITY OF TECHNOLOGY
Faculty of Electronics, Communications and Automation

Markus Aalto

Design and Implementation of Java based Distributed Data Storage

Diplomityö, joka on jätetty opinnäytteenä tarkastettavaksi diplomi-insinöörin tutkintoa varten Espoossa 25.11.2009

Työn valvoja Prof. Heikki Saikkonen

Työn ohjaaja DI Juha Larjomaa

TEKNILLINEN KORKEAKOULU **DIPLOMITYÖN TIIVISTELMÄ**

Tekijä:	Markus Aalto
Työn nimi:	Hajautetun tietovaraston suunnittelu ja toteutus Java-kielellä
Päivämäärä:	25.11.2009 **Sivumäärä:** 76
Tiedekunta:	Elektroniikan, tietoliikenteen ja automaation tiedekunta
Professuuri:	T-106 Ohjelmistotekniikka
Työn valvoja:	Prof. Heikki Saikkonen
Työn ohjaaja:	DI Juha Larjomaa

Palvelukehitysalusta on asiakaskohtaisten palveluiden kehitystä varten luotu ohjelmisto mobiiliverkko-operaattoreille. Alustalla toteutettavat palveluohjelmistot tarjoavat operaattoreille korkean käytettävyyden ja suorituskyvyn, yhdistettynä tulevaisuuden kasvuodotukset mahdollistavaan skaalautuvuuteen. Palvelukehitysalusta asennetaan osaksi operaattorin verkkoa, ja se tarjoaa ympäristön sekä palveluiden luomista että niiden ajamista varten. Tyypillisesti palveluohjelmistot liittyvät useisiin operaattorin järjestelmiin, kuten verkon viestikeskuksiin, palvelutarjoajien sovelluksiin ja business tuki järjestelmiin.

On tavallista, että palveluohjelmistot sekä käyttävät että tallentavat tietoa ohjelman suorituksen yhteydessä. Tallennettava tieto voi olla joko pysyvää, tai tilapäistä ja lyhytaikaisesti säilytettävää. Aiemmin palveluohjelmistoissa tiedon tallennukseen käytettiin korkean käytettävyyden omaavia tietokantoja. Korkean käytettävyyden tietokannat ovat tyypillisesti sekä kalliita että monimutkaisia. Lisäksi tietokannat yleensä vaativat ylimääräistä laitteistoa korkean käytettävyyden saavuttamiseksi.

Tämän diplomityön aiheena on hajautetun tietovaraston suunnittelu ja toteutus. Toteutus on optimoitu tiedon lukemista varten, ja se tarjoaa tiedon pysyvän tallennuksen yhdistettynä korkeaan käytettävyyteen. Tieto hajautetaan järjestelmän kaikkiin solmuihin, ja se tallennetaan jokaisessa solmussa paikallisesti. Tietovarasto toteutetaan komponenttina, joka integroidaan osaksi palvelukehitysalustaa. Komponentti tarjoaa sekä palvelukehitysalustalle että palvelusovelluksille luotettavan tallennuspalvelun klusterissa.

Avainsanat: Java, Hajautettu tietovarasto, Palvelukehitysalusta

HELSINKI UNIVERSITY OF TECHNOLOGY **ABSTRACT OF THE MASTERS THESIS**

Author:	Markus Aalto
Name of the Thesis:	Design and Implementation of Java based Distributed Data Storage
Date:	November 25, 2009 **Number of pages:** 76
Faculty:	Faculty of Electronics, Communications and Automation
Professorship:	T-106 Software Systems
Supervisor:	Prof. Heikki Saikkonen
Instructor:	M.Sc. Juha Larjomaa

Service creation platform is a development platform that is used to create customer specific service applications to operator networks. Service applications must support high availability and high performance with sufficient level of scalability to support future traffic growth. Service creation platform is located in the operator network, and it provides business logic creation and connectivity framework to enable flexible service creation. Service applications typically connect to various operator business support systems, core messaging components and content provider applications.

Service applications almost always need to read and write service execution related persistent or transient data. Previously a highly available database was used for providing such storage services for the cluster of service nodes. However, highly available databases are typically either expensive or complex, and they often require additional hardware support for providing the high availability.

The target of this thesis work is to design and implement a distributed data storage component, which is optimised for read access. The implementation ensures data persistence and high availability using local file system disks and transaction distribution between the cluster nodes. The component is fully integrated into the service creation platform providing the clustered data storage services for the platform itself and the applications built on top of the platform.

Keywords: Service delivery platform, Distributed data storage, Java

Preface

I would like to thank professor Heikki Saikkonen for supervising this thesis work. Also I would like to thank Mikko Kestilä and Sami Katainen for letting me to work on this component, and all the other colleagues at work for providing comments and feedback during development. My deepest gratitude goes to Juha Larjomaa, who provided valuable input and guidance during the process of writing this thesis.

Additionally I would like to thank my family for letting me spend some extra hours on finishing this thesis. Especially I like to thank my wife Hanna-Mari, and our daughters Annika and Jenni for understanding and all the support.

Vantaa 25.11.2009

Markus Aalto

Table of Contents

List of Figures

List of Tables

Abbreviations

2PL	Two-Phase Locking
API	Application Programming Interface
ARC	Adaptive Replacement Cache
COTS	Commercial Off-The-Self
CRM	Customer Relationship Management
FIFO	First In First Out
GGSN	GPRS Gateway Supporting Node
JAIN	Java Advanced Intelligent Network
Java EE	Java Platform, Enterprise Edition
JCS	Java Cache System
JDBC	Java Database Connectivity
JFS	Journaling File System
JSLEE	JAIN Service Logic Execution Environment
JVM	Java Virtual Machine
LRU	Least Recently Used
MM7	MMS content provider (VASP) interface protocol
MMSC	Multimedia Messaging Service Centre
MTBF	Mean Time Between Failures
MTTR	Maximum Time To Repair
NFS	Network File System
O&M	Operations and Management; Operations and Maintenance
ODBC	Java Database Connectivity
PDU	Protocol Data Unit
RAC	Real Application Clusters
RAID	Redundant Array of Inexpensive Disks
SDP	Service Delivery Platform
SNMP	Simple Network Management Protocol
SQL	SeQueL
VASP	Value Added Service Provider

1. Introduction

1.1. Problem Statement

Service applications and middleware running in the operator networks have very strict requirements on system availability, performance and scalability. In order for the software vendor, developing such software products to be competitive, it is not enough that the system meets the mentioned requirements. They also need to be easy to deliver, simple to manage and provide endless possibilities for operator specific customizations. The systems must also have a competitive price.

One possible solution for overcoming the problem is to implement a service creation platform, which supports many of the functionality typically required in the systems delivered to the operators. The platform must support interfaces, which allow customizations of the existing services, flexible integration to the operator systems and quick creation of new service logics to meet the operator demands. The service creation platform should also come without excessive additional license and hardware costs.

The platform and the services built using the platform need to provide a solution for highly available and high performance data storage. As the operators come in different sizes, the storage needs to scale in terms of performance but also in terms of price.

1.2. Goals

The target of this thesis work is to design and implement a distributed data storage component, which can be used to store various application specific data. For the service applications used in the operator networks, the component should provide similar functionality that typically requires use of a highly available database. With no need for highly available database, it will be possible to eliminate the extra hardware, software license and installation costs involved with the external database systems.

The distributed data storage component will be implemented in Java and integrated with the service creation platform. It will be used to create highly available service applications to operator networks.

1.3. Scope

This thesis work concentrates on design of the common data storage component for the highly available service creation platform. It also presents the component requirements from the platform perspective.

This work describes and analyses theory related to the data caching, data distribution and transaction management in order to understand the complexity of the problem and to find solutions to the mentioned problems. Theory part is investigated as a literature study.

Part of this work is the high-level design for the component including the protocols used for transaction distribution and communication between multiple cluster nodes. This thesis work also covers the implementation of the storage parts shown in Figure 5. Implementation related to transaction distribution and lower level communication protocols are not part of this thesis work.

It is not in the scope of this thesis to create a storage implementation for all known storage problems. For many data storing purposes, for example, the relational database will still be the most logical choice. Limitations of the component will be described in the analysis part and in the final conclusions.

The performance results from the component testing have been obtained from a gateway storage replacement project and are not done as part of this thesis work. However, the obtained test results will be shown in the analysis part of this work in order to analyze the success of the implementation against the criteria.

1.4. Structure of the Work

Chapter 2 describes the background information related to the service creation platform, the applications, characteristics and the services it provides. Distributed data storage is part of the service creation platform and this chapter also shows the positioning of the system in the context of the operator.

Chapter 3 introduces the theory related to different kind of storages, caching and transactions. This chapter also contains an analysis of existing Java based components implementing the functionality required to store data with reliability and persistency.

Chapter 4 defines the criteria for the component with use cases and set of requirements. Chapter 5 describes the component architecture in high-level. It also describes the high-level implementation part. Chapter 6 presents the analysis of the work against the criteria.

Finally, chapter 7 presents the conclusions, known limitation of the component and some areas of improvement for the future.

2. Service Creation Platform

2.1. Overview

The service creation platform is a platform creating and delivering customised solutions to telecom operators into SDP domain [2]. The SDP is a framework used for defining the network infrastructure for creating services. The service creation platform is an implementation of service creation environment as defined in the SDP terminology.

Applications in SDP domain are used for providing services, for example, to end-users over mobile or converged networks. They are also used for controlled access point to content and service providers providing secure access to operator network, allowing services to be delivered, targeted and personalized for the subscriber.

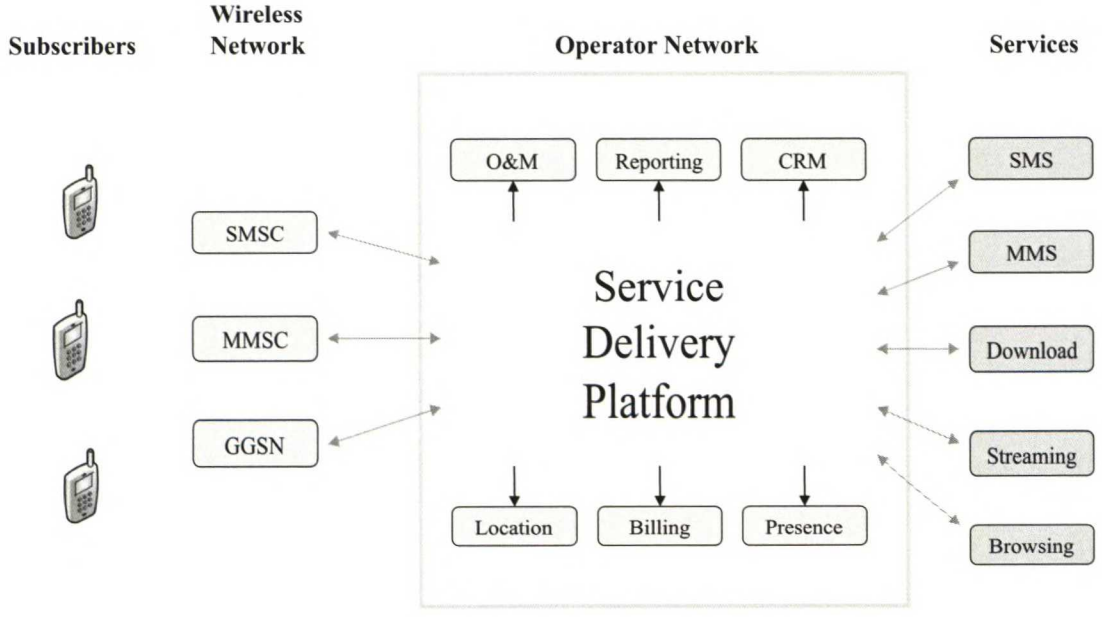

Figure 1 Service delivery platform

As shown in Figure 1, the SDP is part of operator's network. Applications running within the SDP framework integrate with network core components such as GGSN or messaging centres, supporting applications such as CRM, location or billing systems, and third party applications, services and content providers.

2.2. Platform Layers

The purpose of a service creation platform is to increase code reuse, improve quality and increase productivity allowing rapid development of new services into the SDP domain. The service creation platform is shown in Figure 2, dividing the system into multiple layers.

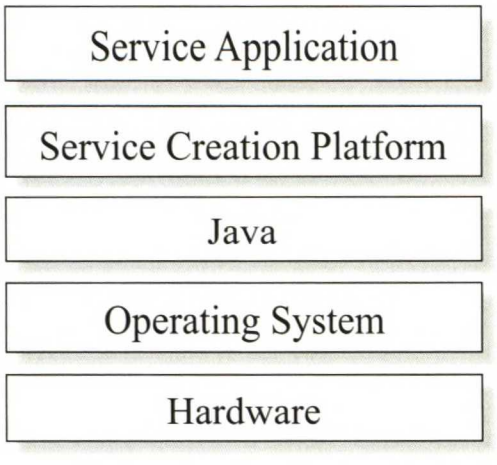

Figure 2 System layers

At the bottom layer of the system is the hardware that is used to physically run the software. Hardware is mainly standardised commercial of-the-self (COTS) rack installed hardware, which has very good price to performance ratio and is typically preferred by operators. Only few hardware platforms are supported to minimize the testing related costs.

On top of the hardware layer is the operating system layer and the services provided by it. This can be for example Sun Solaris or one of the commercially supported Linux versions such as Red Hat Enterprise Linux. The choice of operating system may also have relation with the hardware vendor.

Next to the operating system layer becomes the Java virtual machine implementation for the specific operating system. Java provides the abstraction layer between the hardware and operating system. This allows applications to be run in different hardware and operating system environments.

Service creation platform runs on top of Java Virtual Machine (JVM) and provides the services for the Service Application layer. Service creation platform

provides the functionality commonly required in the SDP service applications. Typically the SDP use cases share a large amount of common functionality, such as O&M services, support for application clustering, external system connectivity and frameworks for creating user interfaces.

Service Application is in the top-most layer. Its implementation is based on the requirements given by the customer. This may be a product or customer specific solution that is tailored for exact customer specific needs. The exact scope and functionality depends on the business needs, and the other network elements the Service Application needs to connect with.

2.3. Platform Characteristics

This chapter defines the characteristics of the service creation platform as considered to be important for the applications in operator networks and in particular in SDP domain.

Following characteristics are important characteristics of the service creation platform:

- High availability
- Low latency
- High performance and scalability
- Extensibility

2.3.1. High availability

When systems and services need to run 24 hours per day and 7 days a week additional consideration needs to be taken care of when designing the system. One important thing to specify is the required level of availability, as the required availability can have a huge impact on the design of the system.

The availability is typically calculated using following equation:

$$\text{Availability} = \text{MTBF} / (\text{MTBF} + \text{MTTR})$$

Equation 1

Where Availability is the degree of availability expressed as a percentage. MTBF is the mean time between failures, and MTTR is the maximum time to repair or resolve a particular problem. 99.999% availability can be reached for example

with system components having MTBF of 100000 hours and MTTR of 1 hour. This would mean that within more than 11 years, the system would have a downtime of one hour.

The design for system availability should cover both planned and unplanned downtime. Design for high available system must consist of the whole system, including the hardware, network, software components and the environment where the system is running. For example, failures in the operating environment, such as failures in cooling systems, could cause system to fail earlier than expected in normal operating environment.

Generally it is impossible to create a system, which is 100% available, so there needs to be a clear target for the availability of the system during the design. The target availability level should be derived from the cost of downtime, and the amount of money to spend on the system [1].

The system typically reaches high availability if the system is designed using component level redundancy, where special failover configurations are used to extend the availability over single server availability.

Redundancy

Redundancy is a technique where components of the system are duplicated (or multiplied in general). At least one of the components can be used at any one time. The system can also continue to function when one of them fails, and typically there will be no impact to the system operation when that happens. This approach is very typical with hardware components where each hardware part in the system is duplicated.

Redundancy enhances reliability, recoverability, serviceability and manageability [3]. Reliability of the system towards the users increases when they do not experience service outages when hardware or software component fails. Recoverability is improved as the error condition can be corrected almost immediately using the redundant component. Serviceability is improved as the failed component can be repaired while the redundant component continues to function, typically even without shutting down the system. Manageability is improved, as there are less service outages, which results into system that is more manageable.

In hardware redundancy configuration, the used hardware components (e.g. servers and storages) are typically equipped with dual power supplies, dual network cards and redundant disk configurations.

In software redundancy configuration, the application is installed into multiple server nodes. Application instances can be running as independent service instances or in a cluster forming a single system.

In environmental redundancy configuration at least two instances of the same system is installed into separate geographical locations.

Clustering

Cluster is a computer system, which consists of multiple servers working as a single entity. Clustering provides the service redundancy by allowing the application to be run in multiple server nodes. Clustered application can be running in either symmetric or asymmetric configuration.

In symmetric failover configuration the system has two or more servers forming a cluster, each server providing service simultaneously. If one server fails, then one of the other servers takes the load of the failed server, or load is distributed between the remaining servers. The load can be evenly distributed between application instances using load-balancer network elements.

In asymmetric failover configuration there is one or more separate standby servers next to the active servers. If one server fails, then one of the standby servers becomes active and takes the load of the failed server. Activation procedure can be either manual or automatic.

Geographical Redundancy

Geographical redundancy is a way to provide resilience over site failures. This form of redundancy is used to protect over catastrophic failures such as earthquakes, which may cause the whole site to fail.

In this setup the systems can be totally isolated from each other, meaning that two or more independent systems are used for providing the service. Independent system can consist of a single server or a cluster of servers.

In a more complex scenario a single system is distributed into multiple geographical locations. In this case any data that is required or generated by the system may need to be replicated between the servers in all of the sites. Complexity of this setup is primarily caused by the increased network latency, lower bandwidth and higher probability of failures over the connections between the sites.

In all of the cases the access to services can be routed through load-balancers with capability to fail-over to the alternative system available to provide the service.

2.3.2. Low latency

Latency is a time delay between the relational events. In communication networks the network latency is the amount of time between sending a packet over network from one computer and to receive it in another computer. One form of latency is the round-trip latency, which is used to measure how long it takes to send a single packet to a system and then receive a response.

If application can send only one packet at a time, then it can send maximum of $1/latency$ packets per seconds. If the round-trip latency over connection is 0.5 seconds, then maximum of 2 packets can be sent per second. Additional latency can be caused by anything introducing additional delay to the flow of events.

As service applications are typically placed between end-users and service providers, it is important that the applications themselves do not cause long latency. It must also be possible to create service applications, having – if not guaranteed maximum latency – at least small average maximum latency. For example, in a smart web proxy application that is used for billing of web browsing traffic towards content services, and is placed between end-user's web browser and service provider's content server, it is important to keep the latency of individual requests at minimum. If individual requests would have too high latency, it would render the user experience of the browsing service unusable.

Importance of low latency becomes critical when a service application needs to provide services to the operator's core network elements having strict

latency requirements. In such an environment, it must be possible to estimate the expected maximum latency caused by the system and control it.

In Java based applications the latency of the system cannot be easily predicted. Automatic memory management (garbage collectors), inadequate scheduling control and unpredictable synchronisation delays [14] are the main reasons for the unpredictable latency. Due to the mentioned limitations, standard Java or Java Enterprise Edition have not been successful in real-time applications that have a strict requirement for predictable latency.

2.3.3. High performance and scalability

Today's commodity hardware can offer very high performance for applications. However, in typical operation services the system performance needs to also improve over time. This means that the system needs to support scalability. Often high performance and scalability do not necessarily go well together. Scalability requires a more complex implementation, which in turn results into lower performance than in a simple implementation. However, this may often be justified, for example in systems, which are required to support a large number of simultaneous users with low response times.

Systems can be said to be capable of "scaling up", when the performance of the system can be improved by improving the performance of a single server. This can be achieved by using faster CPU's or by adding more CPU's. Often scaling up also requires adding more main memory and using faster disks. Scaling up also requires that the tasks performed by the application can be run parallel between the multiple CPU's of the server. Modern operating system assigns work between multiple CPU's automatically, if the applications are running in separate processes. If the services are running within a single process, then the application must split the tasks into multiple parts and use multi-threading. Operating system is then able to assign the tasks using threads between multiple CPU's.

Performance of the system can also be improved by scaling the system horizontally ("scaling out"). Horizontal scaling means that the application is run in multiple servers. Horizontal scaling may also provide high availability as the application is run in multiple servers, and if one of the servers fails, the remaining ones can still provide the service.

Horizontal scaling does not necessarily mean that the system is clustered and all servers are running the same application. An application can also be partitioned to handle different tasks in different servers. In larger systems the partitions can then again be scaled up by using bigger servers or scaled out by adding multiple servers to handle the specific tasks.

Running the application in multiple servers changes the programming model from a single server model as all application data is no longer available in local memory, and the changes to the common data needs to be shared between the application instances running in separate nodes. How the application data needs to be shared depends on the model how the system scales out. If the application is running in a cluster, the data needs to be somehow made available between the nodes. On the other hand, if the application is partitioned, the data may also be partitioned between the individual application partitions. How this is implemented depends on the application and the data shared by the applications, and requires careful planning.

Java environment offers extensive features for creating multi-threaded applications. This, combined with design, which is targeted for parallel execution of services, provides the foundation for scaling up the system by adding more CPU's.

The distributed data storage that is implemented as part of this thesis work provides support for scaling out the application to multiple servers in either cluster or partitioned configurations.

2.3.4. Extensibility

Service applications in the operator networks are running in a very heterogeneous environment. Typically no two systems are equal, for example, in terms of connectivity requirements. This means that it must be possible to easily extend the standard product offering with customer specific connectivity solutions.

The service creation platform provides an extensible connectivity framework, where protocol level functionality is encapsulated into components called connectors. Connectors are integrated to the platform connectivity framework, which provides a common API for the business logic to receive events from and to send events to the external systems.

The business logic framework provides the routing functionality between the connectors and the business logics. This approach allows, for example, the connectors to be changed or new ones to be added to match the customer specific environment without requiring changes to the product application level logic.

In addition to the customer specific connectivity requirements it is common that the operators have specific requirements for the application level business logic, which cannot be satisfied with existing product functionality. The platform therefore also supports easy modification of the application level business logic, by allowing changes to be made to the existing business logics and by adding of completely new application logic overriding and replacing existing logics.

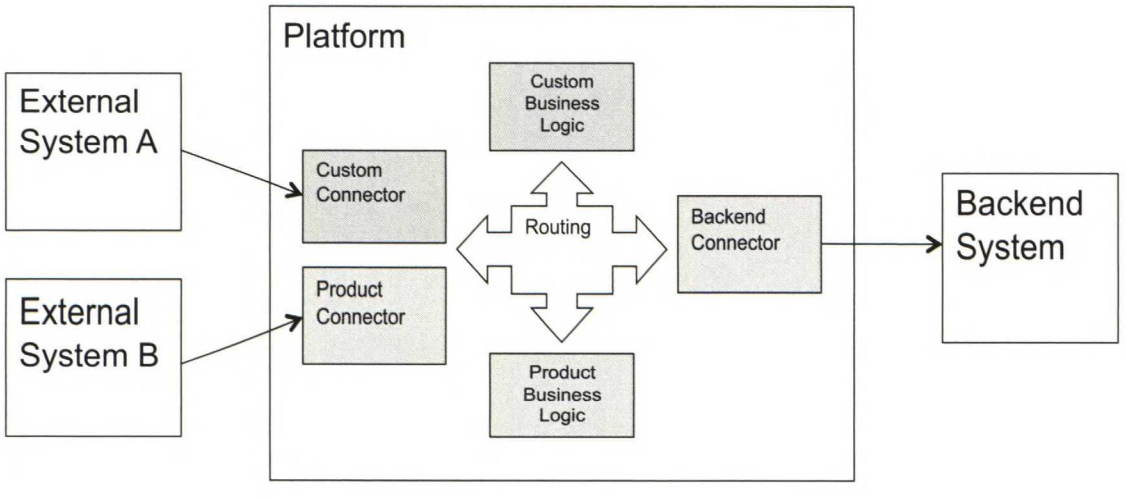

Figure 3 Platform and customisations

Figure 3 shows a simple example application with product business logic and connectors. The product is customised with customer specific business logic and a connector. The Customer Connector, the Product Connector and the Backend (Product) Connector are used for communication to external systems. All connectors are integrated with the platform connectivity framework [1]. The Product Business Logic and the Custom Business Logic are used to create the required business logic. Both business logic components are integrated with the platform business logic framework [2]. The routing functionality between the

[1] Connectivity framework is not shown.
[2] Business logic framework is not shown.

connectors and the business logics is part of the platform. The platform can be configured to make correct routing decisions according to the customer requirements.

2.4. Platform Architecture and Services

The service creation platform provides subsystems, which implement some of the commonly required platform services. Each subsystem provides services through service specific interfaces. Interface API's are used by other subsystems and components to integrate with the services. As subsystem implementation is hidden behind well-defined service interface, the implementation may be extended or modified without changing the service specific interface and affecting other subsystems using the service.

Some of the most important platform services are described in following subchapters.

2.4.1. Extension service

The Extension service provides automatic discovery, instantiation and runtime life cycle management of components developed on top of the platform. All components that are developed on top of the platform should integrate with the Extension service.

Components communicate with each other through service interfaces. Extension service provides access to component implementation using dynamic binding based on the service interface type. This allows component implementations to be easily replaced for customisation purposes.

2.4.2. O&M services

The service creation platform provides a common O&M service framework that can be used for operating, administering, managing and maintaining the system. Service applications and components that are integrated with the platform O&M service framework can utilise the tools provided by the platform, for example, for configuration and monitoring. Some of the most important O&M subsystems are described below.

Provisioning service

The provisioning service is used for creating and managing the configuration of the system – to adapt the system into the specific environment and use case. Provisioning can be done during installation or during normal maintenance operations. Provisioning service can be used, for example, to configure the platform and product components, add new users and roles to the system, control system's access privileges and control the cluster nodes of the system.

Part of the provisioning service is the internal service interfaces, which are used by the application components to publish configurable parameters. Each component can publish its own set of parameters to be controlled by the provisioning service. Provisioning service can control the component's parameters in runtime and it stores the configuration to the persistent storage.

Another part of the provisioning service is the interface, which is used for configuring the system. In the platform the provisioning interface is Web based Provisioning GUI. All platform and service configuration changes can be made using a web browser. Provisioning GUI allows configuring of all components that have properly published their own parameters to the provisioning service. GUI pages are created automatically based on the information available through the internal provisioning service interfaces.

In the platform the provisioning service is implemented by the provisioning subsystem.

Monitoring service

The monitoring service is used for monitoring the platform and the application components. Monitoring means that the user of the system – for example system administrator – can monitor the status and behaviour of the system. Monitoring information can be, for example, external system connection status or traffic counters exposed by a connector for monitoring.

Monitoring is a central part of the O&M functionality of the system. Typically it can be used to verify that the system is functioning as expected, or it can be used to find out problems in some specific area of the system. It can also be used to produce statistical and reporting information, which is based on history of monitored data.

In the platform the monitoring service is implemented by the monitoring subsystem.

The monitoring subsystem implements also support for SNMP (Simple Network Management Protocol) based monitoring. SNMP is a protocol used for monitoring systems and network elements over IP based networks. SNMP provides simple monitoring functionality where the SNMP manager (typically some form of management console) contacts directly the system component (SNMP agent) for providing monitoring information. SNMP only works when there is connectivity between the monitored and monitoring systems.

Alarm service

The alarm service is used for sending alarms from the platform and application components. Alarms are used to notify the user of the system – for example system administrator – about any critical or irregular event in the system.

It can be used to provide information about failures in the system, which requires immediate actions to keep the services online. For example, to notify that one of the cluster's server nodes has failed.

It can also be used for pro-active system maintenance and control, for sending warnings from events possibly requiring further investigations. For example, to notify that there is flow control issue in the system, which may cause problems in future if not resolved in time.

In the platform the alarm service is implemented by the alarm subsystem. The alarm subsystem also implements support for SNMP based traps.

SNMP protocol supports alarm information that can be sent over IP based networks in form of SNMP traps. SNMP traps are events sent from the managed system (SNMP agent) on need basis (immediately when the alarm condition is noticed) to the SNMP manager. SNMP manager can then show the alarm originating from the particular network element allowing the network administrator to react to the event as soon as possible.

Logging service

The logging service is used for providing core event logging services for the platform and the applications. The goal of the logging service is to support

debugging and maintaining the software during development and at production sites (customer installations). The components can use the logging service to write detailed information about the internal execution of the component. Logging services records events in a certain scope in order to provide an execution trail that can be used to diagnose problems in particular area of the system. The scope is defined by the application or component that is writing to the log.

Logging can be used for recording system related information regarding the status of the software component and the tasks that it is executing. It can also be used for recording detailed alarm specific information, generating log records when alarm condition is reached, or to log monitoring information either when monitored value changes or in regular intervals.

2.4.3. Clustering service

The clustering service is mandatory for supporting the scalability, redundancy and high-availability requirements of the system. The clustering service is used to manage the application nodes in a cluster, providing start-up and shutdown functionality as well as constant health monitoring of each cluster node. When one of the nodes in the cluster fails, the failure is automatically detected in remaining cluster nodes by the clustering service.

The clustering service can be used to build applications, which are aware of other applications and services in a cluster. For example, it can be used to build automatic fail-over in application level where one application node assumes the tasks processed by one of the failed cluster nodes.

2.4.4. Connectivity service

The connectivity service provides a framework for creating connectors. Connectors can be used both to receive and send messages, and they can be part of the product or the customer specific service application. A connector encapsulates the protocol specific interface implementation, which is used for communication towards the external systems.

When connector receives a message in the protocol level, it converts the message to an internal event. The connectivity framework reads the event from connector and passes it to the business logic framework.

When business logic needs to send data, the business logic framework passes an event to the connectivity framework that routes the event to the correct connector. Event is then converted to protocol message and sent out.

Connector implementations may use existing higher-level API's for implementing the protocol (e.g. Java Mail API for SMTP protocol) support in the connector. Alternatively the protocol can be implemented in protocol data unit (PDU) level, for example, using TCP/IP sockets directly as PDU transport mechanism.

2.4.5. Business logic service

The business logic service provides the framework for supporting service creation. It can be used to bind together the connectors and the business logic in order to create the service specific message flow.

Business logic components are configured to the business logic framework configuration. The configuration defines the static routing for the events from connector to the business logic. Business logic can also route events between business logics in order to support business logic based dynamic routing and business logic layering.

Business logic may access directly the subsystems and other extension components that are created for implementing the product or customer specific services (see chapter 2.4.1 for the details of extension service).

2.4.6. State distribution service

The state distribution service is used by the business logic and subsystem components to store and distribute data between cluster nodes. State distribution service is implemented by the distributed data storage subsystem, which is the result of this thesis work.

For example, in web GUI's user session related data is stored in the server side. The HTTP requests used by the browser client contain only session id information that can be used to identify the session. The user session related data could be stored into the distributed data storage using the session id as a key. By storing the data to the distributed data storage, it would be possible to distribute the

server side GUI logic between multiple cluster nodes as the stored session data would be available for service logic instances running in any cluster node.

2.5. Existing Standards

The SDP applications and platforms are not standardised and there is no single framework implementation to follow for creating applications or platforms within SDP domain. However, there are some existing initiatives and implementations, which are being used in when creating services to SDP domain using Java.

JSLEE (Java Service Logic Execution Environment) and the JAIN (Java Advanced Intelligent Network) community initiative [22] defines a set of Java technology APIs that enable the rapid development of Java based communications products and services using Java. JSLEE focuses on delivering available, reliable and scalable services that are portable across JSLEE application servers.

Java EE (Java Platform, Enterprise Edition) [23] is the industry standard for developing portable, robust, scalable and secure server-side Java applications. Java EE is more targeted to enterprise applications, but many of the technologies and solutions provided by the Java EE environment are also directly applicable to the SDP domain. There are also initiatives, which try to combine some of the JSLEE features into Java EE world.

2.6. Conclusions

At the time when the service creation platform development project was started, the JSLEE and JAIN initiative did not have any real implementations available. Also the Java EE was considered too complex and slow for creating near real-time applications with low-latency and high-performance.

The purpose for JSLEE and the JAIN initiative is identical to the service creation platform described in this chapter. Also the approach selected for the implementation is very similar. This implies that there is real need for a service creation environment, which enables rapid development of high available and scalable service applications into the service creation space.

The Service creation platform uses Java and its Standard Edition version [13]. Standard Java virtual machine implementations do not directly support, for

example, implementations with predictable and low-latency. Some of the design constraints introduced in chapter 4.4 try to minimise, for example, the long garbage collection delays caused by large JVM heap memory configurations. The design and implementation related to the constraints is described in chapter 5.

3. High Available Data Storage

High available data storage is ideally a data store, which is always available. The techniques for storing, caching and managing data in a highly available manner are numerous and widely used, for example, in databases.

The following subchapters introduce some background theory for understanding the

- Techniques used for storing data
- Issues related to data caching
- Managing concurrency using transactions
- Issues caused by data distribution between multiple nodes

3.1. Storage Techniques

Typically a persistent storage is a disk-based system located locally in the computer, shared in a network or distributed in a network of computers (or nodes).

3.1.1. Local storage

Local storage consists of a single hard disk or multiple local hard disks accessed from the locally running applications. The operating system supported file system is used to manage the files. Redundancy can be implemented in the operating system level or in the hardware level, using RAID for transparently mirroring and striping the data into multiple disks.

In some applications local storage may also be implemented as in-memory storage. In-memory storage is very efficient and it is becoming cheaper as the memory sizes are growing and the memory prices are going down. Typically in-memory storages are not persistent, so they can be used only as high performance runtime storage to temporary data.

Local storage can provide very cost-efficient store with high performance for locally accessed data. However, local storage provides no scalability and no real high availability against node failure. Thus it cannot be used as high available storage for multiple nodes unless combined with some form of data distribution.

3.1.2. Shared storage

Shared storage can be, for example, a database accessed using a relational query language (such as SQL) over database connection protocol (e.g. JDBC or ODBC), or a shared network file system accessed as disk storage (SAN or NAS). In shared storage model the storage system itself can be redundant and highly available, and data does not depend on availability of individual client nodes.

If the shared storage is implemented using shared databases, then the database acts as a common storage for the application instances providing the transactional consistency and often also the high availability. High availability in databases is typically implemented either using some form of data replication or distribution technique, or using shared disk storages in redundant configuration.

The shared storage provides easier scalability for the client nodes than local storage. However, it introduces additional latency to the data access as the storage is accessed over the network (typically either Ethernet or fibre channel) connections. It may also become a problem when the shared storage needs to scale for higher performance. Shared storage can also get very expensive when combined with very strict high availability and high performance requirements.

Caching can be used to increase the performance of the shared storage and to reduce the latency introduced by the additional network access. Chapter 3.2 contains more information about caching.

3.1.3. Distributed storage

Distributed storages are used for systems requiring high availability, reliability and scalability. Data is distributed using some distribution technique to a cluster consisting of multiple nodes. Distributed storages are used for systems that are required to provide scalability for large amounts of data, high availability over node failures, and high performance with hundreds or thousands of simultaneous clients.

Data distribution introduces additional problems when scaling the system into multiple nodes. One major issue is the data consistency when updating data in

any of the nodes. Another issue is the additional synchronisation traffic caused by the data distribution between the nodes. Synchronisation also introduces additional latency for data storage operations.

3.2. Caching

3.2.1. Cache overview

Cache can be used to increase the performance of the system when access to the data storage is slow, and the performance of the service is limited by slow storage access.

Caches [19] are used in different ways and for different purposes in computer systems. For example, in computers the performance is increased using smaller and faster memory caches closer to the CPU than the larger main memory. In operating systems the file system uses the disk storage for storing data. File systems use the computer main memory as a cache to improve the file system performance, as the main memory is much faster than disk. Also in operating systems the network file systems use the local disk and local memory for caching network file system data. Although caches are used differently and in different kinds of technologies, they are mostly based on similar concepts. Accessed data entries are stored locally in case of access is made to a remote system, or to memory in case of storage access is made from disk. The idea is to store some of the data into smaller - but faster - storage area in order to speed up data access to frequently accessed data.

Caching increases performance of the system primarily due to following reasons:

- Access of the cached data entries is more efficient allowing the application to serve requests faster.
- Access of the cached data does not cause any load to the actual storage, which can lead to better performance for those data requests that still need to access storage directly.

Caching can also decrease performance. When caching is used each request has an extra cost caused by the search from the cache and the management (storing and removing) of the cache entries. If the requests are too randomly distributed, the storage will be accessed with every request and caching will decrease the system performance.

Figure 4 shows a very simple example of memory cache having a pool of four (4) entries and storage of 10 entries. Each cache entry has a data value containing a copy of some value from the actual storage entry. Each entry also contains a tag, which identifies the entry of the actual storage. In the example this is the address of the storage entry.

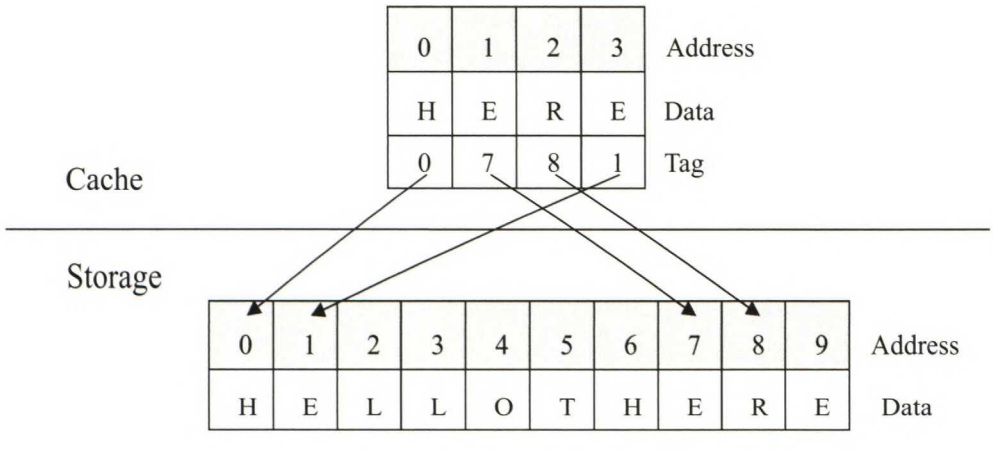

Figure 4 Storage with a cache

When client or application reads from the storage it first makes a lookup from the cache using the address of the data entry. If the data entry is available in the cache (an entry with a tag value matching the request address is found), it can be returned immediately to the client. If the entry is not found, then the value needs to be read from the storage. At this point one existing cache entry is removed and the new read data entry is stored in place.

The decision of which entries are removed from the cache in favour of new read or written data depends on the replacement algorithm (see chapter 3.2.3 'Replacement algorithms' for details).

The way the cache works in terms of writes depends on the cache write policy (see chapter 3.2.2 'Write policy' for details).

3.2.2. Write policy

Write policy controls how data is written to the storage when data is updated [24]. Following write policies are commonly used:

- Write-through policy. Every write to the cache causes writing also to the storage.
- Write-back policy. Writes do not immediately update the storage.

Write-through policy is typically simpler to implement as the data is written immediately to the storage. However, if cached data entries are modified frequently, then write-though policy does not improve the update performance of the system at all.

Write-back policy improves also the update performance, as the cached entry can be written to the storage only when it is required to be replaced with some other entry. The replacement algorithm is used to control the actual replacement logic.

It is also possible that cache allocation is not done for data entries that are written to the storage. The cache is then primarily updated by read access only. Additionally the cache then needs to be updated when the entry, which is already stored to the cache, is updated.

3.2.3. Replacement algorithms

Cache replacement algorithms are a compromise between hit-rate, cost of managing the cache and benefit of accessing the cache over slower storage.

Hit-rate indicates how often an item that is searched for is found from the cache. Higher hit rate typically means better performance, as more search results can be satisfied from the faster cache.

Large number of different replacement algorithms has been implemented for various caching purposes. Some of the simplest ones are Least Recently Used (LRU), and First In, First Out (FIFO).

LRU based cache keeps track of the least recently used cache entries, for example, in a linked list. When entry in the cache is accessed it is moved to the top of the list. When entry needs to be replaced, the entry in the bottom of the list can be replaced. Benefit of LRU is that finding a replacement does not require additional search, however it has the cost of maintaining the list. LRU has also multiple variants of the algorithm (e.g. LRU-K and ARC) with attempt to simplify and improve the implementation.

FIFO based cache keeps track of the age of cache entries. When entry needs to be replaced the oldest one is replaced. Benefit of FIFO is that it is lightweight and simple to implement. Unfortunately it performs poorly in most cache applications.

3.2.4. Cache coherence

A cache system is coherent if, whenever an entry is read, the returned value is the one most recently written [20]. Cache coherency is easy to achieve, if storage is accessed only from one location.

Caching can cause issues with data coherence, when storage is updated and read from multiple locations. In a distributed system the cache can be kept up to date with updates, if storage write triggers a notification event that is sent to all caches. Notification can either invalidate the cache entry or alternatively the notification can contain the updated data value.

3.3. Transactions

A transaction is a single execution of a program [4]. When two programs are allowed to read and write value of the same data item concurrently, the execution should be managed using transactions to prevent data inconsistencies. A program could be used for reserving a ticket in airline reservation system. If multiple reservations could be executed without transactions, it would be possible to reserve same seat numbers twice.

For example, transactions are used in databases, which support simultaneous access to data for multiple users.

3.3.1. Transaction properties

Transaction properties define how the system processes transactions. They define the key properties of a reliable system [7].

Atomic

Transaction is atomic if either all parts of the transaction succeed or none of them succeed. This means that in a transaction consisting of multiple operations, failure in any of the operations cause the whole transaction to fail.

Consistency

Transaction is consistent if the state change caused by the transaction is valid. This means that if the consistency rules of the database or the storage are violated, then the whole transaction will be rolled back and the state is restored back to the state before the transaction was started.

Isolation

Transaction isolation ensures that even if the transactions are executed concurrently they must not access or see data, which is still in intermediate state.

See chapter 3.3.2 'Isolation levels' for detailed information about the typical isolation levels.

Durability

Durability guarantees that when user of the transaction has been notified of successful completion of the transaction, then the state change is made persistent. At this point, the change must also survive system failure.

These four transaction properties are typically called using the ACID acronym. Although, for example, databases implement the ACID properties, they may also introduce some level of relaxation in order to allow better performance or easier implementation. In most databases isolation level can be controlled in transaction level. Also some databases allow less strict implementation of durability properties, and may allow losing some latest committed transactions in case of system or disk failures.

3.3.2. Isolation levels

Transaction isolation levels control how the transactions see the modifications executed concurrently by transactions.

SQL standard [6] suggests the following definition for the transaction isolation: 'The isolation level of an SQL-transaction defines the degree to which the operations on SQL-data or schemas in that SQL-transaction are affected by the effects of and can affect operations on SQL-data or schemas in concurrent SQL-transactions'.

Following transaction isolation levels are suggested in the SQL standard:

Isolation Level	Description
READ UNCOMMITTED	Allows transaction (A) to read data that may be in the middle of being modified by another transaction (B).
READ COMMITTED	Transaction (A) can read only data that has been committed, but it allows reading of changes committed by another transaction (B) for previously read data values.
REPEATABLE READ	Transaction (A) can read only data that has been committed, and any read data cannot be changed before the transaction (A) has committed.
SERIALIZABLE	Transactions occur in a completely isolated fashion; i.e., as if all transactions in the system had executed serially, one after the other.

Table 1 Transaction isolation levels

Phantom reads can happen in all mentioned isolation levels, except in the case of SERIALIZABLE isolation. Phantom read happens if same search condition is executed multiple times inside a transaction, and the resulting set differs between the executions. This may happen when using range searches.

The isolation level that is being used in the transactions indicates the kind of phenomena that can occur during the execution of concurrent transactions. Following phenomena are possible:

Phenomena	Description
P1 "Dirty read"	Transaction T1 modifies value of data object A. Transaction T2 then reads data object A value before T1 commits the transaction. If T1 then performs rollback for the transaction, T2 will have read a value that was never committed and that may thus be considered to

	have never existed.
P2 "Non-repeatable read"	Transaction T1 reads value of data object A. Transaction T2 then modifies or deletes data object A and commits the transaction. If T1 then attempts to reread the value of data object A, it may receive the modified object or discover that the data object has been deleted.
P3 "Phantom"	Transaction T1 reads the set of data objects N that satisfy some <search condition>. Transaction T2 then executes SQL-statements that generate one or more data objects that satisfy the <search condition> used by the transaction T1. If transaction T1 then repeats the initial read with the same <search condition>, it obtains a different collection of rows.

Table 2 Possible transaction phenomena

In addition to the isolation levels suggested by the SQL standard [6], there is also a SNAPSHOT isolation level. In SNAPSHOT isolation level the transactions seem like they would be executed in their own snapshot of the database without seeing other transactions [25]. SNAPSHOT isolation is weaker isolation level than the SERIALIZABLE level. However, due to the implementation being closely related with the implementation of multiversion concurrency control, several database vendors have adopted it as one of the supported isolation levels.

3.3.3. Concurrency control

Concurrency control is used for managing the execution of concurrent transactions in a system while supporting the mentioned transaction properties. Concurrency control is used for ensuring the serialisation of transactions and consistency of data.

Concurrency control can be implemented using a number of different algorithms. Typical concurrency control algorithms are based either on locking or on timestamp based transaction ordering [8].

Locking

Lock based concurrency control algorithms work so that the transactions access resources using read and write locks. Lock granularity for a resource in a database or storage can be, for example, in data item level. When transaction needs to read data, it first acquires a read lock for the resource after which it can proceed to read. When transaction needs to update data, it first acquires a write lock for the resource after which it can proceed to update the data.

Read locks are used to provide shared access to the data, which means that multiple readers can access the data concurrently. Write locks provide exclusive access to the data, which prevents concurrent read and write access from other transactions. If lock cannot be acquired immediately the transaction must wait for the lock to become available before the transaction can proceed.

Two-Phase Locking (2PL) and its variants are the most common algorithms for implementing locking based concurrency control [8, 4, 11].

Basic 2PL consists of 2 phases:

- Phase 1 is the locking phase. Locks are acquired on all items that are accessed, and no locks are released.
- Phase 2 is the unlocking phase. Locks are released on all items and no locks are acquired.

Phase 1 ends when processing related to the transaction has ended and no more locks are acquired. During phase 1 and phase 2, the data items can be read and updated. Phase 2 ends when all locks have been released.

Advantage of locking based concurrency control is that it is reasonable simple to implement and it guarantees the transaction serialisation order. Disadvantage of locking based algorithms are that the concurrency and performance may be affected, if multiple transactions need to access same resources concurrently. In case the transactions are long lived the performance may be severely affected.

Timestamp ordering

Concurrency control algorithms using timestamp ordering are based on controlling the transaction serialisation order using transaction and resource specific timestamps [4]. When transaction starts, a unique timestamp is created for a transaction. Transaction timestamps must be generated so that the relative order of timestamps corresponds to the order, in which the transactions initiated.

Each resource item that is accessed by transactions keeps resource specific read- and write-timestamps. Read-timestamp is the highest timestamp of any transaction that has read the item, and write-timestamp is the highest timestamp of any transaction to have written the item. Scheduler uses the transaction and resource timestamps to ensure the correct transaction order. If the order of access would result into incorrect serialisation order, then the conflicting transaction is aborted and restarted with new timestamp.

Basic timestamp ordering (Basic T/O) and strict timestamp ordering algorithms are common implementations of the timestamp ordering based concurrency control.

Multiversioning

Multiversioning is a technique used, for example, by Oracle databases for transaction concurrency control. In multiversioning multiple versions of the same data is kept in the storage when data is updated [25, 26]. Old versions are cleaned up by the system when the transactions terminate. Multiversioning does not require locking, but can also be applied with locking based algorithms

Optimistic concurrency control

Optimistic concurrency control (OCC) algorithm has similarities with timestamp ordering based concurrency control algorithms and does not require locking [26]. OCC is optimistic by assuming that the transactions typically do not perform conflicting modifications. If they do, the conflicts are resolved by restarting transactions.

OCC provides most benefit when data access is mostly reading, or when writes only affect small portion of the total data.

Deadlocks

Deadlocks are typical when concurrency control is implemented using locking algorithms. Transactions may get into deadlock due to two transactions X and Y waiting for a lock for the some resource, as it is possible that the transaction X already has acquired a lock to the resource requested by transaction Y, and at the same time transaction Y has already acquired a lock to the resource requested by transaction X.

Deadlocks can be prevented by ensuring that deadlocks never happen by designing the system appropriately, by avoiding the deadlocks using runtime decisions or by detecting the deadlocks in runtime [10].

3.3.4. Failures & Recovery

Failures may be caused by software and hardware failures, but when the system crashes it causes transactions in the middle of processing to fail. It is important for the system to be able to recover the already committed transactions and ignore the not committed transactions [4].

In order to survive from software crash or failures caused by loss of power, the data must be stored into persistent storage, which can survive over system restarts. Typically the system disk storage can be used as persistent storage. There are number of ways the storage and the recovery from failures can be implemented, but a common way is to write all transaction related updates into change log or journal file, which is then written to the disk. The data storage itself containing the current values may fit fully into the main memory, or is partially in disk storage and partially in main memory.

The change log file is used for recovery from failures. The updates are stored into change log as sequence of entries. The change log entry may contain at least following information:

- Transaction identifier. Used to identify the transaction.
- Item identifier. Used to identify the updated item.
- Value. Used to store the new value for the updated item.

In addition to the change log entries, there should be transaction related entries, which indicate the start of the transaction, commit of the transaction or abort of the transaction.

When failure happens and recovery is needed, the change log is scanned to find all the committed transactions. All transactions, which were not committed to the change log, may be ignored. All committed transactions are redone to the storage, and when all transactions have been redone the storage has been fully recovered.

Even the above does not protect from media failures or from fire. Protection from storage media failures can be done, for example, using redundant disk configurations, writing duplicate change log file to alternative disk, or even archiving the change logs to alternative location.

3.3.5. Transaction Distribution

The chapter 3.3.3 described the concurrency control techniques for managing the transaction concurrency in a system where there is only one copy of each data item. In distributed environment where multiple copies of each data item are distributed to multiple nodes the locally applied concurrency control is no longer sufficient for ensuring the serialisability and atomicity of the transactions [4].

Replication

In distributed environment multiple copies of the same item are stored. The distribution of data items can be done in multiple ways. The exact choice of replication strategy depends on the requirements of the storage.

- High availability for each data item can be reached by replicating all items into minimum of 2 nodes. The distribution may be based on, for example, some consistent hash algorithm.
- Maximum performance for data that is mostly read can be reached by distributing the data items to all nodes. However, distributing data to all nodes limits the system's scalability for writes.

If the distribution of data is asymmetric, the implementation becomes more complex as it is necessary for the system to locate the nodes containing copies

of the data. Also any node failure can cause excessive data transfer between the nodes. The latter is to ensure the high availability of each data item.

The replication strategy for writes may be either synchronous (Eager) or asynchronous (Lazy). With synchronous replication writes are applied over all copies of data as part of the distributed transaction. Use of synchronous replication ensures the consistency of the data when combined with distributed concurrency control method.

With asynchronous replication one copy is written as part of the transaction, and other copies are written asynchronously. Although lazy replication may cause data consistency and transaction isolation issues, it is widely used as it offers better scalability. In some applications – like in web page caches – it may not matter if the data that is read is not the latest version, as long as it is immediately available for reading and can be updated almost immediately.

Concurrency control

Distributed transactions can be considered as a group of transactions consisting of multiple sub-transactions running in separate nodes. Because of this, the transactions in distributed environment can be mostly handled using similar techniques that are used for concurrency control in single server environment. Most popular techniques are based either on distributed locking or timestamp based protocols.

Table 3 lists some of the approaches that are available for locking based implementations:

Method	Description
Write-Locks-All	Write locks are obtained for all copies of the item A during write. Read lock may be obtained from any copy of item A during read.
Majority	Write locks are obtained for majority of the copies of item A during write. Read locks are obtained for majority of the copies of item A during read.
Primary Copy	One node (primary node) manages the locks for item A. All lock requests to the item A must go through that node.

	Different nodes can manage locks for different items.
Central Node	One node manages the locks for all items. All lock requests must go through the central node.

Table 3 Methods for distributed concurrency control

Each method described in Table 3 has differences regarding the number of control messages needed to write or read, and regarding recovery handling. Write-Lock-All and Primary Copy methods are efficient when most of the access is reading.

Distributed commitment

Distributed concurrency control is not enough for solving some of the problems, which are present in distributed environment. In distributed environment it is possible that any node participating or coordinating the transactions fails at any time, or that there are communication failures between the nodes.

The distributed commitment process ensures the atomicity of the transactions and recovery from failures. Commonly used approach for managing the distributed commitment is the Two-Phase Commit protocol [4].

In Two-Phase Commit protocol coordinator node controls the transaction and the other nodes are called participants. The protocol contains two separate phases: voting and decision phase. During voting the coordinator and the participants vote whether the sub-transaction in each node may commit or if it has to abort. This information is provided back to the coordinator, which then based on responses to all sub-transactions, decides whether the transaction commits or aborts. Two-Phase Commit protocol uses timeout-based mechanism for detecting failures in the nodes participating to the transaction.

Two-Phase Commit is a blocking protocol, meaning that there is a possibility to reach a condition where the transaction may block. Blocking means that the sub-transaction in one of the participating node does not know whether the transaction should be committed or aborted. There is also an alternative protocol called Three-Phase Commit, which has been implemented to reduce the likelihood of blocking over Two-Phase Commit.

3.4. Java Based Distributed Storage Implementations

This chapter contains an analysis of existing Java based distributed storage or cache components. The analysis is done based on the information available at home pages of each component.

Components are analysed using following criteria:

- Transaction support
- Storage size support
- Support for Java objects
- Licensing, required 3rd party components and libraries
- Any additional features

3.4.1. Java Caching System

Java Cache System (JCS) is a Java based cache, which is mainly designed for local caching using memory or disk [15]. JCS supports distribution using specific plug-ins. It also supports storing data into SQL database as blob data.

JCS has no support for transactions, and it may encounter data consistency and isolation issues in distributed environments. It supports Java objects, and is targeted for high performance read-only applications. It provides most benefit for applications, which need to cache data that does not change frequently.

JCS is a subproject of Apache Jakarta project, and uses the Apache open source licensing model. It has very limited dependencies to other components and can be run using Java Standard Edition. Architecturally JCS uses a plug-in model, where new plug-ins can be developed to support, for example, different kinds of storage and distribution mechanisms.

3.4.2. Ehcache

Ehcache is a widely used java distributed cache for general-purpose caching, Java EE and lightweight containers [16]. It supports local memory and disk based storages, but it also supports various distribution techniques such as synchronous and asynchronous replication, and replication by copying or invalidating cache items.

Disk based storage can also be made persistent. Ehcache should allow large cache sizes of up to several gigabytes. Ehcache is strictly a cache as it automatically removes data entries from the storage, if it runs out storage space.

Ehcache is not transactional although it supports synchronous replication. Even with synchronous replication it is still possible to run into data consistency and isolation related issues. It supports Java objects, and is targeted for large, highly concurrent systems. It provides most benefit for applications, which are mainly reading from the storage. For example, web pages can be cached efficiently with Ehcache.

Ehcache is using Apache 2.0 open source licensing model. It has some dependencies to other open source components and can be run using Java Standard Edition.

Ehcache can be extended using a plug-in model, where new plug-ins can be developed, for example, to support loading data to the cache and replicating data between caches.

3.4.3. JBoss Cache

JBoss Cache is also a widely used cache and comes as part of the JBoss Application Server. JBoss Cache supports transactions, cache eviction rules and loaders with multiple replication strategies [17]. Replication can be either synchronous or asynchronous, having both Copy and Invalidation based replication supported as well. In addition to above, JBoss Cache supports a replication technique called buddy replication, which allows replication only to limited set of nodes in the cluster.

Although the JBoss Cache name says it is a cache, it can be used as a persistent storage using cache loaders. JBoss Cache supports multiple cache loaders, which can be used to persist data to disk or to database.

JBoss Cache may also run in transactional mode ensuring consistency of the data between concurrent transactions. Version 3 and later supports both READ_COMMITTED and REPEATABLE_READ isolation levels using an implementation of multiversion concurrency control scheme. It also supports

Java objects, and it is targeted for high available and fault tolerant applications.

JBoss Cache is licensed under LGPL [30] and OSI-approved open source license. It has minor dependencies to some other open source components and can be used with Java Standard Edition.

JBoss Cache is no longer actively developed for new features, and new feature development will be done in Infinispan project [18]. Infinispan is new distributed data storage, intended to be an evolution version of the JBoss Cache.

3.4.4. Memcached

Memcached is a high performance, distributed memory object caching system that can be used, for example, to speed up web server applications. It does not support transactions, and it runs as a separate daemon process (not made in Java), so it is therefore used over a separate API. Multiple API's are available for several languages such as C/C++, PHP and Java. It also supports Java objects through the API.

Memcached instances can be run in multiple nodes. All data in the cache is stored to the cluster of nodes using internal distribution logic. However, Memcached is not a storage. It is a cache, which sits between the application and, for example, the database. The application needs to store the data into the cache, and remove the data from the cache. It is primarily meant for caching of large amounts of data, offloading work from the database, which is used as primary storage.

Memcached is licensed under BSD OSI-approved open source license [30]. It has dependency to libevent event notification library, which is licensed under 3-clause BSD license. It works directly on top of Linux, BSD or Windows operating system.

3.5. Summary

The theory in the area of transactions, storages, transaction distribution and concurrency control algorithms has been well covered in the last 30 years. This is mainly due to the work done in the database developments and lately also in distributed scalable storages. However, there is no single standard and best implementation available to make the design and implementation decision.

The multiversion concurrency control is widely supported by the databases and some distributed storage implementations, but it also comes with additional complexity.

The area of existing implementations is very interesting. There are few very promising components available, which could be used in place of the distributed data storage that is implemented in this thesis work. When the original service creation platform project started, the situation was not as promising. The implementation done as part of this thesis work also has some unique approaches to some of the problems faced in many customer installations. One of the approaches is described in chapter 4.4 with more detail.

The techniques presented in this chapter will be utilised in the design and implementation part. Selection of the appropriate algorithms will be based primarily on the requirements set for the component (see chapter 4 for the Criteria). In terms of balancing between design decisions, the target is to try to favour simplicity over complexity.

4. Criteria

The target criteria define the requirements for the distributed data storage component. Requirements are captured in terms of use cases [5], additional functional requirements, non-functional requirements and design constraints.

4.1. Use Cases

Use cases are often used during software development for requirements capturing. They are used to define the needs of the users as they use the system. The users are defined in terms of actors. Actors do not need to represent human users, but they can also be other systems or external hardware that interact with the system.

Use case descriptions are used between the developers and the customer to define in a non-technical way the behaviour of the developed system. They can be described using natural language (e.g. English) and do not require the customer to learn complex notation for describing a use case.

4.1.1. Actors

Based on the analysis the following actors were identified for the distributed data storage:

- Application. The software application, which needs to store application specific information to the distributed data storage in reliable manner.
- Admin. The administrator of the system, who may need to change the system configuration or monitor the health of the system.

Additionally some failure specific use cases were also identified.

4.1.2. Application initiated use cases

In this high-level use case scenario the application using the distributed data storage uses the component API to initialize the access to the storage. After the API and storage initialization has completed the application can perform one of the presented use cases.

Application reads from the storage

Application running in one of the cluster nodes reads data from the storage. The application can read any data stored to the storage transparently regardless, which node has stored the data. The storage provides the stored data items as Java objects.

Application may expect the data be accessible and consistent even if application instance running in another cluster node is modifying the data simultaneously.

It shall be possible for the application to identify the situation where the storage does not contain requested data.

Application writes to the storage

Application running in one of the cluster nodes writes data to the storage. Write operation may be an update of existing value, or write of new value. Written values shall be available for reading by any instance of the same application running in the cluster.

It shall be possible to store the data persistently, so that the stored data survives over node and cluster wide restarts, or from node failures.

Application deletes from the storage

Application running in one of the cluster nodes deletes data from the storage. Delete operation removes existing data, and ignores removal of non-existing data. Removed data should not be available for reading by another instance of the same application running in other cluster nodes after the delete operation has been completed.

Application searches from the storage

Application running in one node needs to search some specific data from the storage. It shall be possible for the application to create application specific search logic or indexing structure to satisfy the requirements of the application logic.

Application initializes the storage from external data source

During system start-up it shall be possible for the application to use some external data source such as file or database to initialize the contents of the storage.

In addition, it shall be possible for the application to keep the external data source consistent and up to date with the updates made to the storage.

4.1.3. Administrator initiated use cases

In this high level use case scenario the system administrator operates the cluster system having the distributed data storage component.

Administrator changes the distributed data storage configuration

It shall be possible for the administrator to use the service creation platform provisioning tools to configure the data storage's configuration.

Administrator monitors the status of the distributed data storage

It shall be possible for the administrator to use the monitoring tools provided by the service creation platform to monitor the status of the data storage. At least following information shall be available:

- Status of the data storage instance in the cluster node.
- Size of the data storage
- Number of data items currently stored to the data storage

Administrator adds a new node to the cluster

It shall be possible for the administrator to configure a totally new node to existing running cluster without restarts.

Administrator removes an existing node from the cluster

It shall be possible for the administrator to configure and remove an existing cluster node from the running cluster without shutting down the remaining cluster nodes.

Removing of an active (or running) node shall cause automatic and controlled shutdown of the removed node.

Administrator activates a cluster node

It shall be possible for the administrator to activate (or start) a cluster node, which is currently inactive. The node may be inactive, as it has been recently added to the cluster, it has been temporarily shutdown or it has failed.

While the new node starts up it will join to the cluster currently formed by the running nodes. All the existing data storage content will be made available for the applications running in the new node as required in chapter 4.1.2. Additionally all the data storage updates made by the applications running in the started node will be made available for the applications in the original cluster.

All transactions started in the original cluster nodes shall be completed successfully during the start-up process.

Administrator shuts down a node from the cluster

It shall be possible for the administrator to shut down a running node from the cluster without affecting the remaining cluster.

While the node is shut down the existing data storage content shall be fully available for the applications running in the remaining cluster nodes as required in chapter 4.1.2.

All the transactions committed in the shutdown node will be completed successfully. All transactions started in the remaining cluster nodes will be completed successfully.

4.1.4. Failure use cases

Service application crashes

This use case covers the uncontrolled shutdown of the application due to application crash.

When the service application fails the existing data storage content shall be fully available for the applications running in the remaining cluster nodes as required in chapter 4.1.2.

All the transactions already committed in the failed application will be completed successfully. All transactions not committed in the failed application will be aborted. All transactions started in applications running in the remaining cluster nodes will be completed successfully.

Server node has a failure

This use case covers the uncontrolled shutdown of the node due to node hardware failure or operating system crash.

When the service node fails the existing data storage content shall be fully available for the applications running in the remaining cluster nodes as required in chapter 4.1.2.

All the transactions already committed in the failed node will be completed successfully. All transactions not committed in the failed node will be aborted. All transactions started in the remaining cluster nodes will be completed successfully.

4.2. Additional Functional Requirements

This chapter elaborates the additional functional requirements for the component.

Additional functional requirements are divided into three high level requirements:

- Storage requirements
- Transaction distribution requirements
- Scalability requirements

4.2.1. Storage requirements

This chapter explains in detail the requirements related to the storage.

Support for multiple application data types

The distributed data storage shall support application specific data types. It shall be possible to create data objects using Java primitives or objects composed of Java primitives. It shall be possible to store data into arrays of objects and common Java collections classes.

High available storage

The distributed data storage shall provide high available storage using data redundancy. Data cannot be lost in case of any single point of failure in the cluster. Data shall be available for reading and writing from other server nodes in the cluster.

Concurrent storage

The distributed data storage shall support multiple concurrent operations in a single server node and in a cluster of multiple server nodes. It shall be possible for multiple different components to use the distributed data storage without being aware of other users.

Transactional access

The distributed data storage shall support transactions with following minimum set of properties:

- Transactions shall be atomic. All updates related to the transaction must succeed, or none of the updates succeed.
- Committed transactions shall leave the storage into consistent state.
- It shall not be possible for a transaction to read data that has not been committed.

Local caching

It shall be possible to have a local caching for improving the read performance for frequently accessed data. Locality of access ensures lower latency and reduces the network traffic and overall load in the cluster.

Persistent storage

It shall be possible to store the changes committed by a storage transaction into durable storage. The storage contents shall be automatically recovered after system (cluster wide) restarts.

System shall support recovery with automatic detection so that only the last valid data in the cluster is recovered.

Persistent storage shall allow backing up of the storage data, and recovery using backed up data.

4.2.2. State distribution requirements

This chapter explains the requirements related to the state distribution.

Transparent distribution

The distributed data storage component shall distribute the data into the cluster of nodes transparently without the application having to implement anything regarding the distribution. When new nodes are added to the cluster or existing nodes are removed from the cluster, there should be no need to change the client application or its configuration.

The decision regarding the distribution must be automatic based on the cluster configuration and the actual running nodes of the cluster.

Transactional distribution

Transactional distribution shall be used in order to ensure that the data storage in each node will have consistent view to the data. State distribution is successful only if it can be completed successfully in all active participating cluster nodes.

When participating node fails during transaction distribution, the failed node shall be removed from the distribution unless failed node was the coordinating node.

When coordinating node fails during transaction distribution, the remaining nodes shall discover the status of the transaction from other participating nodes.

4.2.3. Scalability requirements

This chapter covers the requirements related to the scalability.

Read optimised scalability

The distributed data storage shall be optimised for read access.

Alternative distribution models

It shall be possible to support alternative scalability models without requiring changes to the existing applications.

4.3. Non-Functional Requirements

This chapter covers the non-functional requirements.

Integrated

Data storage implementation shall be fully integrated with the service creation platform in order to provide consistent user experience for the system user, and to benefit from the O&M functions provided by the platform.

It shall be possible for the service creation platform components such as provisioning subsystem to use the data storage component for cluster-wide data storing and distribution.

Extensible

It shall be possible to extend the basic data storage API to support application specific requirements, such as indexing, searching and automatic data expiration rules.

Performance

An application using the data storage will have at least 10 times the performance compared to an application using a highly available external database with similar hardware, when data access profile is mostly reading (> 95%).

An application using the data storage will have roughly identical performance compared to an application using highly available external database with similar hardware, when data access profile is mostly writing (> 50%).

4.4. Design Constraints

This chapter goes through some of the design constraint set for the distributed data storage component.

All data is serialized in the storage

When data entries are stored into the area of the data storage that is in JVM heap memory, the stored objects shall be serialized and written into large byte arrays.

Following approach allows large amounts of data objects to be stored into JVM heap memory without causing excessive garbage collection delays.

5. Design and Implementation

5.1. High-Level Architecture Overview

High-level architecture describes the main components of the storage, including the external components that are using or are used by the storage.

The storage is split into components as shown in Figure 5.

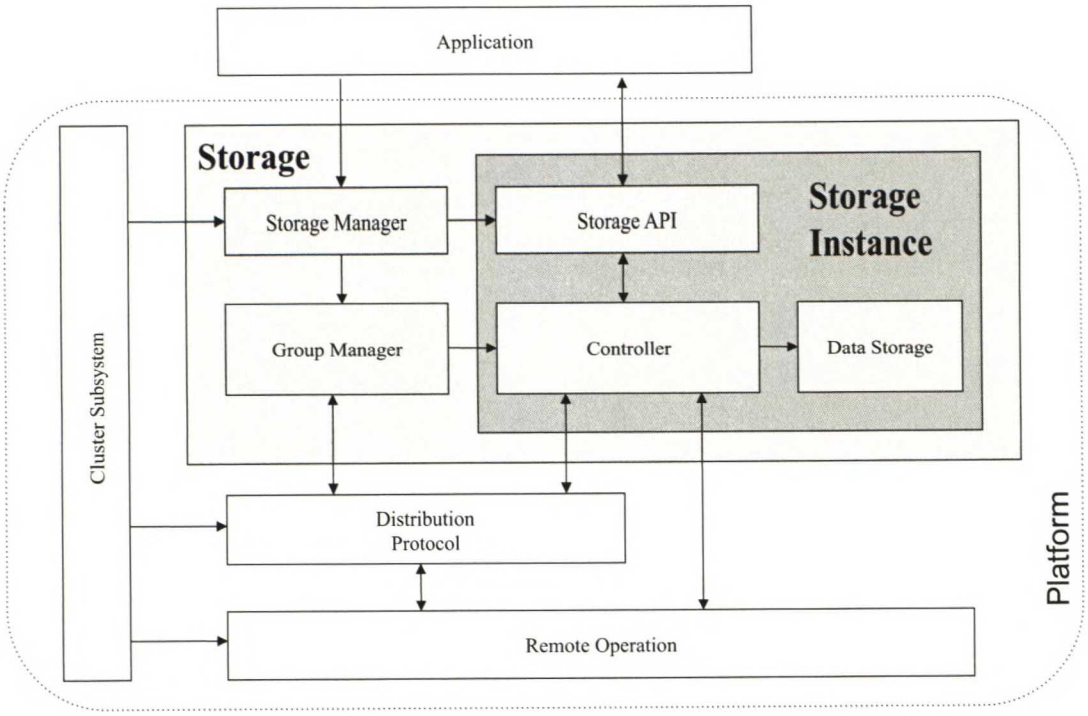

Figure 5 High-level architecture diagram

The coloured parts of the figure above are within the scope of this work. The parts, which have no colour, are essential parts of the complete solution, but are not covered by the storage implementation. Everything inside the rectangle surrounded using a dashed line belongs to the service creation platform implementation.

Application

Application part is the implementation of the service running on top of the service creation platform and uses the data storage. It is not part of the storage implementation itself, but is the user of the storage. From storage perspective multiple applications using the storage can co-exist in one system. Each

application typically creates one or more storage instances using the storage manager API's.

Storage Manager

Applications can create new storage instances in runtime. Storage manager implements the functionality for managing multiple storage instances. Storage manager creates an instance of the controller according to the runtime configuration and binds it with the storage API. Once the local controller instance has been created and the API binding has completed, it registers the storage instance to the group manager, which manages the storage instances in the cluster.

Each system node has one storage manager instance running and it provides for the application the local service interfaces for creating the storages.

Storage manager is also storage's local integration point to the cluster subsystem. All cluster layout changes (adding of new nodes to the cluster configuration, removing of nodes or node crashes) are notified to the storage manager. Storage manager dispatches the changes in the cluster layout also to the group manager.

Group Manager

Group manager implements the functionality for managing the storage instances in a cluster.

Main responsibilities of the group manager are:

- Joining of local controller instances to the cluster.
- Removing of local controller instances from the cluster.
- Shares storage controller instance information between all the nodes in the cluster.
- Informs the local storage controller instances when cluster layout changes.
- Informs the status changes in corresponding storage group controller instances in the cluster.

Group manager uses the distribution protocol component for sharing the storage controller instance information in the cluster. Use of distribution protocol ensures that updates to the cluster storage group's view in each cluster node is atomic and consistent.

Storage API

Storage API implements the API used by the developer for accessing the storage. Each storage controller instance has one instance of the Storage API.

Storage API uses the controller to access the storage, and to receive notifications from storage changes. Notifications can be used in the Storage API to maintain internal data structures, or in the application side to trigger application specific logic. One real world example of using notifications is to manage up to date search index for supporting application specific searching.

Current implementation supports also new implementations of Storage API's as well as extending of existing API implementation.

Controller

Controller implements the node specific logic for controlling the start-up, shutdown and transaction logic of a storage instance. The control logic is implemented as state machine, and the states are described in more detail in chapter 5.2.5.

Controller uses the data storage component for storing and accessing the data locally. Read-only transactions are executed directly using the local data storage. Write transactions – both locally and remotely initiated ones – always go through the distribution protocol component.

Remote operation component is used for special peer-to-peer communication tasks with other cluster nodes. This happens primarily during starting state, when the storage contents are synchronised between the cluster nodes. The start-up storage synchronisation is mainly batch data transfer between two storage controllers, which doesn't need to be atomic in nature.

Local application originated transactions access the controller through storage API.

Data Storage

The data storage component implements an API with methods that allow the transactions to be executed into local storage. Both read-only and write transactions are supported in the API. Storage controller uses the data storage API for executing the transactions.

In addition to the basic transaction execution, the data storage API provides support for creating a storage snapshot using content iterator. The content iterator is used to iterate through whole storage contents. While the iterator is open, all changes to the data storage are only written to the journal file. With the content iterator it is possible to provide a consistent snapshot of the data in the storage, for example, when synchronising the storage contents to the starting cluster node.

Internally the data storage component is divided into following smaller components:

- Search index; manages the references to current data items that are stored to the local data storage.
- Journal log writer; stores the transaction journal for the transactions accessing the data storage.
- Local lock manager; manages transaction read and write locks for the local data storage instance.
- Memory manager; manages the free memory allocation for the local data storage instance.
- Block storage; file or heap memory based storage to read and write data items. Data items are stored using consecutive data blocks. Memory manager manages the free blocks for the block storage.
- Heap memory cache; manages caching of data entries using heap memory.

Cluster Subsystem

Cluster subsystem is a service creation platform core component that provides services for managing the nodes forming a cluster. It is used to start and stop individual nodes, and detect failures in the cluster.

Cluster subsystem also provides services for the service creation platform components such as the storage component using the cluster notification API. When cluster subsystem detects a node failure (either software or hardware failure) it uses the notification API to notify that the cluster layout has changed. In the storage component it means that the group manager then re-arranges the transaction distribution layout and starts new transactions using the modified layout.

Distribution Protocol

Distribution protocol component provides an implementation of distributed commitment protocol. The implementation is based on Two-Phase Commit protocol introduced in chapter 3.3.5.

Distribution protocol uses the remote operation platform component, which implements the low-level cluster communication protocol.

Distribution protocol integrates with the cluster subsystem and listens for cluster subsystem's cluster layout notifications. It also automatically reacts to the changes in the cluster layout, which would affect either currently executed transactions or new transactions. By quickly adapting to the cluster layout changes, the recovery of the transactions can be started immediately when the cluster subsystem reports node failures.

Remote Operation

Remote operation allows a component running in a node to perform low-level operations remotely in remote nodes. Basically it allows for a service operation provider in the cluster to register an operation listener for receiving messages from other cluster nodes, and for sending responses to requests. Service operation provider can also send peer-to-peer messages and multicast messages to other service operation provider counterparts in the cluster.

Remote operation implements the low-level messaging functionality, which is used for communicating between the nodes in the cluster. It abstracts the low-level protocol implementation from the higher-level components. It also allows different protocol implementations without changes in the upper layer components such as the distribution protocol. The current protocol implementation is using point-to-point TCP/IP connections.

Remote operation is listening for cluster subsystem's cluster layout notifications and automatically reacts to changes in the cluster layout. It automatically sets up or tears down the TCP/IP connections between the nodes in the cluster. It also implements a fail-fast mechanism to send immediate responses to message sending requests, which are targeted to the removed (or failed) nodes.

5.2. Implementation

5.2.1. Storage model

The storage model provided by the distributed data storage component is based on use of local disk or memory using transaction distribution to replicate changes between the cluster nodes. Each node in the system uses an API that is used to create access to the storage. Multiple groups of storage instances can be created to isolate the storage areas between applications.

During storage creation the group name specifies the storage group to be accessed. If the storage does not find an existing storage group using the group name, it will automatically create a new storage group. If existing storage group is found, then access to the existing storage group is provided.

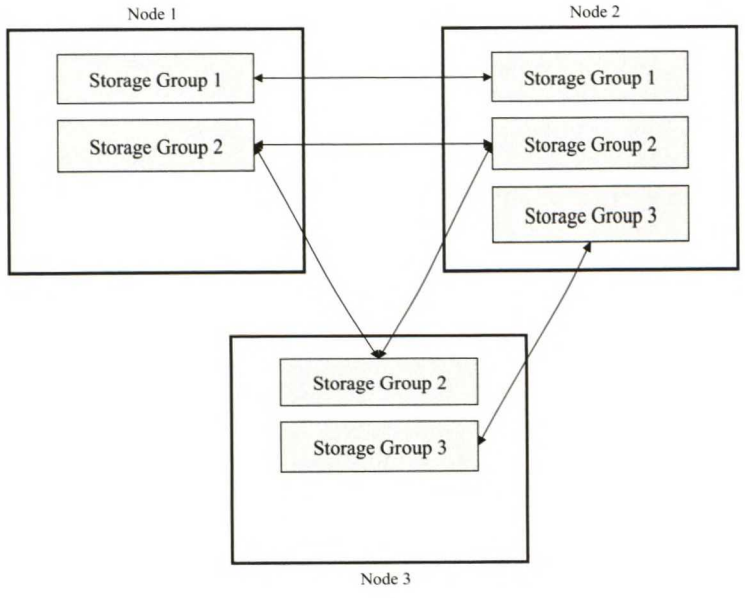

Figure 6 Distributed storage groups

Figure 6 shows an example distribution of storage groups in a system of three nodes, having three separate storage groups.

- Storage Group 1 contents are distributed with nodes 1 and 2 only.
- Storage Group 2 contents are distributed between all the nodes.
- Storage Group 3 contents are distributed between nodes 2 and 3.

Single service application may create one or multiple storage groups, but it is also possible for multiple applications to share a single storage group. Purpose of having multiple storage groups is to isolate the application specific storages from each other, and allow possibility for tuning and optimisation of a storage group based on the access profile and stored data.

The storage uses the same JVM process memory as the service creation platform, sharing the memory between different service applications.

Local storages

Each node has a single JVM process and the processes share the common data using the storage. Locally in each node the storage area is implemented either using local file system and disks, or using the internal heap memory of the process. The exact implementation can be selected per storage group configuration.

Local file system based implementation stores all data from single storage group to file system files. Files are used to store all the data in single storage group to ensure local access to all data, but it can also be used to create storage sizes larger than the available heap memory.

Local heap memory based implementation stores all data from single storage group to large byte array allocated from the JVM heap. Heap memory based storage groups are therefore limited by the amount of heap memory allocated for the JVM.

Memory management

Storage's internal free space management is based on binary buddy system algorithm [20]. The binary buddy system allocates storage's free space into blocks whose sizes are specified in powers of 2. The implementation of binary buddy system can be used to get an address to a free segment of memory consisting of consecutive blocks. When allocated data entries are freed, they

are released back to the buddy system, which then manages joining of neighbour blocks again into larger blocks.

Storage contents

Storage contents are accessed through the storage API. Content data items are accessed using a key, and stored as key – value pairs.

Key identifies uniquely the data entry within the storage group. Key is a combination of Java class type and its object content. Keys are identical only if they are of same Java class type with equal serialised content.

Keys and values are managed as Java objects in the storage API. Internally in the storage the key and value objects are always serialised. Key is stored into internal index, which is used to hold reference (address) to the data item in the storage. Data entries are always searched from the internal index using the key.

Serialised key and value objects are stored together to the local storage in serialised form. Keeping the objects in serialised form allows them to be distributed between the nodes without any extra serialisation and de-serialisation overhead. Storing both the key and the value objects together allows the storage recovery to find the valid key and value pairs just by scanning the storage content sequentially.

Storage persistency

Storage groups can be created in persistent mode. Persistency ensures that the data is stored to non-volatile storage, and that it can be backed up and recovered in case of system wide restart. In heap memory based configuration there is a background process, which ensures that all data stored to the heap memory based storage is also flushed to the disk. This configuration ensures faster performance when smaller storage sizes are required.

In case of persistent storage groups the distributed storage component makes sure that only the storage group instance with latest valid data (master instance) can be started. Other instances need to wait until the master instance does the data recovery from persistent storage and becomes available for sharing the data.

Large storage sizes

The modern 64-bit JVM implementations allow specifying JVM heap sizes of several gigabytes. Although it is possible to have a large JVM heap, it can severely affect the garbage collection times and system latency in particular when objects have long lifetimes.

The implementation tries to solve this problem by serializing the objects and storing them into large byte arrays. Data elements are stored as objects inside the JVM only when they are in application control ensuring that even large amount of objects in the storage or the size of the objects do not severely affect the JVM garbage collection times.

As the implementation may also store all data to the file system, the maximum storage size is theoretically limited by the amount of available disk space.

5.2.2. Local cache

As default the local data storages are configured to use file system files as data storage. In such a configuration the storage can also be configured with heap memory based cache on top of the file system.

All access to local storages goes through the local cache (if available), which ensures that the cache is coherent with the storage at all times. When entries are accessed they are first searched from cache. Local storage will be accessed only if the entry is not available in the cache. All entries read from the local storage are also placed to the cache.

Heap memory based cache uses write-back policy for controlling how the writes are applied to the storage. LRU algorithm (Least Recently Used) is used for the cache replacement policy.

5.2.3. Transaction model

In the storage implementation all transactions are atomic in nature. All transactions also try to ensure consistency of the storage.

There are three main types of transactions supported by the Storage API:

- Read-only transaction, which can be used to read the local storage.

- Write transaction, which writes to the storage without application control.
- Write transaction, which writes to the storage with application control.

The application control feature allows the application to control the behaviour of the operation at state when data entry specific write locks have been successfully acquired from the storage.

Read-only transactions

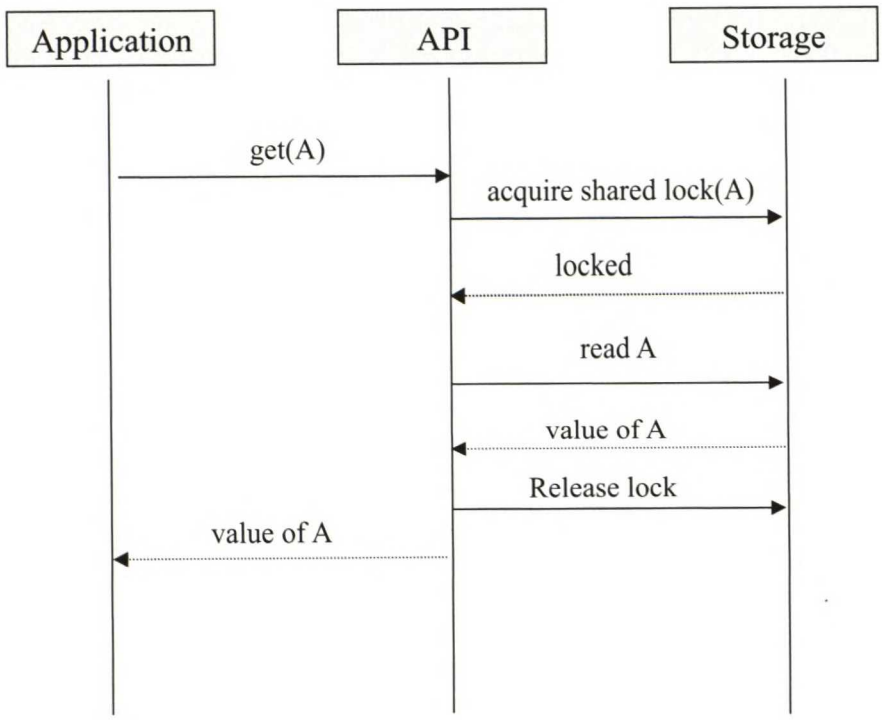

Figure 7 Get-operation sequence

Read-only transactions are implemented using an API method call also referred as get-operation. When application wishes to read from the storage, it will call a method in Java API with array of keys. Once all data entries have been locked, the values for each entry will be read, locks will be released and the values are returned to the caller. Figure 7 shows a successful read-only transaction reading the value of item A using the get-operation.

Write transactions

Write transactions are implemented using single API method call in a same way as get-operations. When application wishes to access the storage it will call a method in Java API stating the list of entries to insert, update or delete. In

successful transaction the method returns normally when changes have been committed. In transaction failures the API throws a Java exception, storage ensures consistency and does rollback automatically.

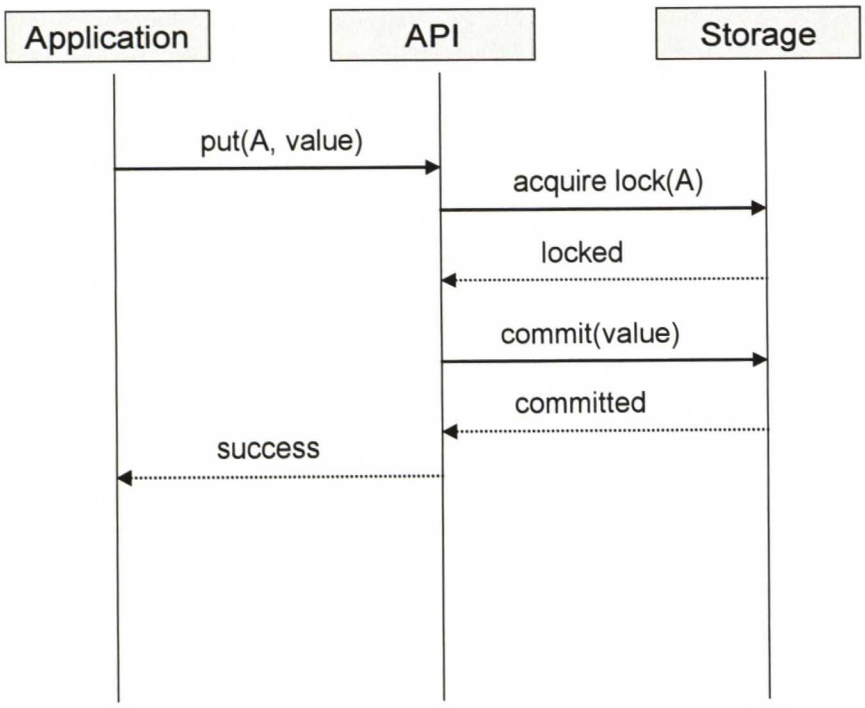

Figure 8 Put-operation sequence

Figure 8 shows a successful write transaction (put-operation) without application control. The transaction is initiated by the application and the figure shows the corresponding sequence flow between the application, storage API and storage.

In addition to simple write transactions, the storage also implements write transactions, which allow application control logic to be run between the transaction start and commit. Application control logic is executed after the targeted storage entries have been successfully locked for writing. With application control logic in place it is safe to read the existing value and make the decision whether entry will be changed, what will be the new value for the entry or whether the transaction can be cancelled. This feature can be used to make transactional updates to the data based on existing value of the item.

Concurrency control

Storage instances use a local implementation of the scheduler for transaction scheduling. Implementation is based on Strict Two-Phase Locking (2PL)

algorithm. The algorithm was selected because there was no need to support long running transactions, it is simple to implement and performs reasonably well for mixed read and write access profiles. The Strict 2PL algorithm implementation uses shared locks and exclusive locks for controlling the access to storage entries at item level. It also acquires all the required locks for the transaction in the start, and releases them all only when the transaction either commits or aborts.

Shared locks are used by read-only transactions and can be shared between multiple transactions. When shared lock is acquired by a transaction, it blocks exclusive lock requests until shared locks have been released.

Exclusive locks are used for all write transactions and cannot be shared with other transactions. When exclusive lock is acquired by a transaction, it blocks all other lock requests until the lock has been released. All write transactions acquire the locks using exclusive locks initially thus there is never need to upgrade the read locks to exclusive locks.

The use of Strict Two-Phase Locking ensures that the transaction sequence is serialisable. Due to the implementation of the transactions with the Strict Two-Phase Locking concurrency control, the storage in practise works in the equal level of SERIALIZABLE isolation level.

Deadlock management

Storage implementation provides simple deadlock prevention between transactions trying to access the same storage data items. The deadlock prevention algorithm is implemented by enforcing the storage entry access order to be the same between all transactions. This is possible, as the Java API requires that the keys for all the data items to be accessed must be provided in the method call. The storage sorts the provided storage entry keys before executing the transaction and thus makes it impossible for two transactions to acquire locks for the entries in different order.

In addition to the deadlock prevention described above each lock attempt will timeout after a lock timeout period. In case acquiring the lock times out the transaction will be retried later. The transaction may also fail if the transaction cannot successfully acquire all locks within the transaction timeout period.

5.2.4. Transaction distribution

Transactions are distributed for all nodes in the cluster having an instance of storage group (symmetric distribution). This approach allows the application to limit the number of nodes the data is distributed in the cluster, while still allowing local access to the data they require. The storage group manager knows the layout of the storage group in the cluster and enforces the system to adapt to layout changes in runtime.

In the current implementation the transaction distribution is always synchronous using the Write-Locks-All method for distributed concurrency control. This approach was chosen primarily to ensure the storage consistency. If the system is configured with heap based memory cache the sub-transactions write through the storage cache, allowing the cache to invalidate or replace the data content with new value within the transaction. This approach keeps the remote memory caches coherent with changes in the storage group.

All the transaction distribution is made in the control of Two-Phase Commit protocol. Use of Two-Phase Commit ensures that all nodes in storage group have consistent view on the storage data. Node and connection failures are detected by the service creation platform's cluster subsystem component and informed immediately to the distributed storage, which can then take appropriate recovery actions.

5.2.5. Storage runtime states

The local storage controllers running in each system node use different storage states for supporting the runtime processing. Figure 9 shows a diagram containing the different runtime states and the possible transitions between the states.

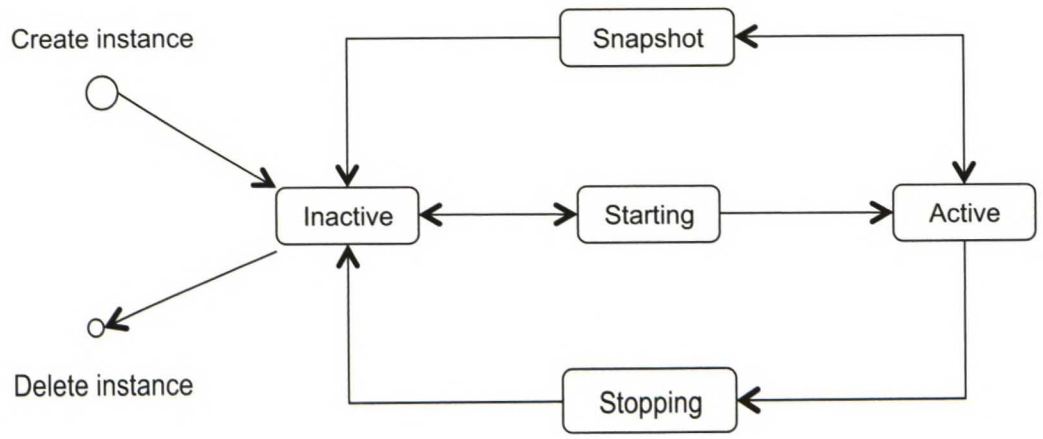

Figure 9 Runtime states

Runtime states are used to control the behaviour of the system in different stages of the storage controller instance. The functionality implemented by each state is described below.

Inactive state

The local storage controller enters into *Inactive* state, when the controller is initialized. This typically happens when the server node is started and the application running in the service creation platform gets started.

Controller may also enter the *Inactive* state after initialization, if an unexpected error is encountered in the storage in runtime, the storage instance is shutdown normally or the start fails.

Starting state

The local storage controller enters into one of the *Starting* states after the local cache controller passes the initialization steps and is ready to join to other storage controller instances running in the cluster. The *Starting* state contains three (3) sub-states, which are used based on the storage controller runtime configuration and the decision made in the previous states.

Initially during starting the controller enters in *Starting Discovery* sub-state. In this sub-state the controller discovers the cluster layout for the storage controller instances in the cluster. The cluster layout controls how the start-up will proceed for the starting instance. At this point, the controller instance may enter either to

Starting Loader sub-state, *Starting Synchronisation* sub-state or directly into *Active* state depending on the result of the discovery process.

In *Starting Loader* sub-state the controller instance may use an external initialization loader to pre-populate the storage. The storage may be pre-populated, for example, from the external database during system-wide restart. After the loader has finished, the storage controller can enter the *Active* state.

In *Starting Synchronisation* sub-state the controller instance synchronises the contents of the local instance during start-up from one of the *Active* instances running in the cluster. This is implemented using the Remote Operation component, which is used to synchronise the data items from any cluster node having the storage controller in *Active* state.

In reality the decision logic is more complex than described above. For example, the logic in *Starting* state needs to consider things like number of instances in the cluster and runtime states of all other instances.

Active-state

The local storage controller enters *Active* state when its storage contents are consistent with other instances in the cluster. At this stage storage controller also starts accepting transactions from local applications, as it is able to serve the transactions locally.

Important part of the functionality provided by the controller in the Active-state is to provide start-up synchronisation services for remote controller instances entering the *Starting* state. During synchronisation the *Active* controller moves the local data storage into a state (*Snapshot* state), which enables replication of the local storage contents to remote controller, while ensuring data consistency on local data storage. The synchronisation is managed on background while allowing new transactions to be processed as normal.

The local storage typically stays in *Active* state until the system is shutdown. When shutdown starts the storage controller enters *Stopping* state.

Stopping state

During normal shutdown procedure the storage controller enters *Stopping* state, which does controlled shutdown of the local storage controller. The shutdown procedure consists of following steps:

- Disconnect the local controller instance from the remaining cluster. No new remote transactions are handled once the disconnecting has been completed.
- Complete the already started local transactions. No new locally started transactions are allowed from this point forward.
- Shutdown the data storage and ensure that in case persistent storage is used the data storage is fully stored to disk.

Snapshot state

The local storage controller can enter *Snapshot* state when requested by the system. This state would allow backups of the storage files to be taken while system is online. In *Snapshot* state the controller instance allows new local and remote transactions, but is not eligible for providing the start-up synchronisation services for remote controllers.

6. Analysis

6.1. Use case analysis

This chapter describes actual real world application examples, which are using the distributed storage component. The application examples are used as part of the analysis to show how the distributed data component can meet the use case targets specified in chapter 4.1.

6.1.1. Existing implementations

Platform configuration data

The configuration data for the service creation platform and for the applications is stored to the distributed data storage. When a node of a system is initially started, the configuration data is uploaded to the provisioning storage from external persistent storage such as database or file system file. External persistent storage can be used to store the initial master copy of the provisioning data. External persistent storage does not need to be highly available, as long as it is backed up on regular basis if there are changes in the master copy.

Figure 10 shows the storage architecture for the platform provisioning data. The external storage is an instance of MySQL database and provides the master copy data source for the provisioning data. Only the Management Node accesses the external storage directly.

Management Node also provides the GUI for provisioning the platform and the applications. Any change made from the GUI will be stored into the external storage and the provisioning storage. Changes will be made available to applications running in Processing Nodes using the replication provided by the distributed data storage.

Alternatively data can be manipulated dynamically in runtime by the components integrated with the platform provisioning subsystem. Components can be running either in Processing Node or in Management Node. In this case, the data modification will be detected by the Provisioning system running in Management Node and stored to the external storage. Any change that cannot be applied to the external storage will be stored in the distributed data storage and removed once the external storage comes available.

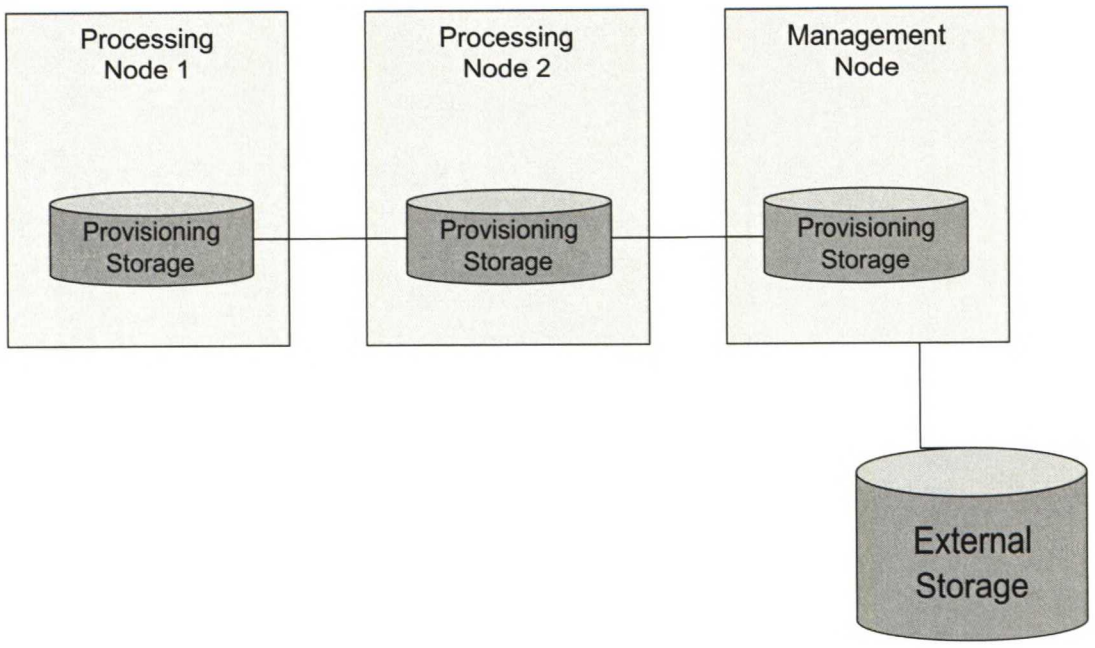

Figure 10 Service creation platform's provisioning storage architecture

The distributed storage provides highly available storage for the provisioning data. All nodes have same view to the data, and due to the symmetric data distribution provided by the storage, the provisioning data will be available in all running cluster nodes. The system will survive even if some of the configured cluster nodes would have a failure, node would be shutdown or external database storage would be offline.

Distributed data storage as gateway message storage

In the in-house developed MMS gateway product a highly available database was used as a persistent storage for storing multimedia message data. The gateway delivery was expensive due to hardware and license cost, but also due to extremely complicated installation process. Development and support for the database also requires a lot of product expertise, which needs to be constantly updated. At the end of 2008, it was decided that the database should be removed, and the distributed data storage component was identified as a primary candidate for the new storage. Primary reason for the replacement project was to remove the extra cost related to software licenses, but also the possibility to remove some of the extra hardware related costs.

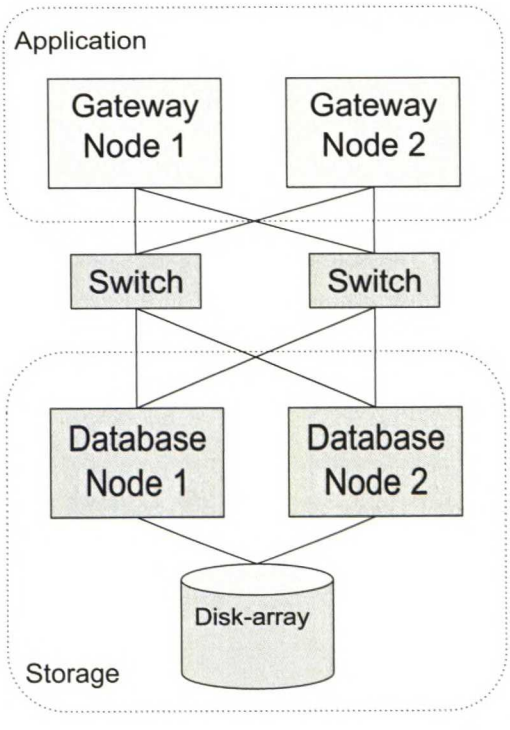

Figure 11 Old MMS gateway setup

Figure 11 shows the deployment architecture of the old MMS gateway implementation using the external database. The database was based on Oracle 10g RAC (Real Application Clusters) product, and was typically installed into two (2) dual CPU server nodes. From each node there was fibre channel connections to external disk-array, which was used as a shared persistent storage for the database. Disk-array was internally redundant for high availability [3]. Gateway nodes in the application layer were connected to the database using redundant network connections through two (2) switches. Both database nodes were using similar network setup.

The message data that was stored into the database consisted of MMS message content, the sender and recipient specific address information and message related status data. The content size of a single message is typically around 100KB.

The replacement project was initially started with a prototype exercise. The target was to create a prototype of the new MMS gateway product using the

[3] In some configurations two external disk-arrays were used with cluster file system replicating data between the disk-arrays.

distributed data storage component. During prototyping the new implementation was able to reach the initial performance and reliability targets. So the replacement project was continued.

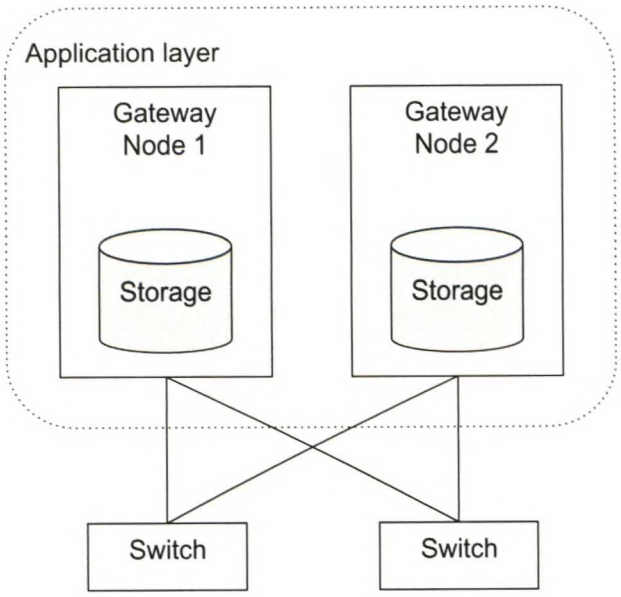

Figure 12 New MMS gateway setup

Figure 12 shows the new MMS gateway setup, which is substantially simpler than the old gateway deployment setup. The new system is less expensive in terms of license and hardware cost. In addition, it is easier to deliver mainly due to reduced hardware complexity.

Based on functional testing the distributed data storage provides exactly the same functionality as the database version of the product. Some changes are required in the JVM and the software configuration, mainly due to now keeping the storage indexes in the same JVM with the gateway application. The system start-up times are now also longer if the message storage is already filled up with large amount of content. This is, however, justified due to the fact that the local disks are used to store data, and the storage performs a data recovery process during application start-up.

The performance testing use case was based on application originated single-recipient message submission over MM7 protocol and delivery to the MMSC over MM7 protocol. This use case was run with both gateway versions.

The hardware configuration used in performance testing was following:

- Old gateway; two application nodes (2 x dual core 3Ghz CPU's) and two database nodes (4 x quad core 3Ghz CPU's). External disk array as shared database storage. Figure 11 shows a picture of the hardware setup.
- New gateway; two application nodes (2 * dual core 2.8Ghz CPU's). Data stored into internal disks. Figure 12 shows a picture of the hardware setup.

Both systems utilise similar network redundancy model having redundant network interfaces and switches, with automatic fail-over between interfaces. Load-balancers are used to balance the incoming traffic between the nodes.

Unfortunately the results are not run with equivalent hardware nodes for the applications. However, setup is close enough to make reasonable analysis.

Gateway version	Performance scale	Performance %	Content size
Old gateway	1	100%	50k
New gateway	0.83	83%	50k

Table 4 Comparison of gateway performance test results

Full performance testing of the new gateway has not yet been finished. Table 4 contains early comparison of the performance test results between the old gateway and the new gateway versions from single use case only. The performance is normalised to the old gateway performance.

The smaller performance number for the new gateway performance means also lower performance in the new gateway. However, the amount of hardware used in the new gateway is significantly smaller. It is also expected that by "scaling up" the hardware, the new gateway should reach similar or even substantially better performance than the old gateway.

6.1.2. Application use cases

Application initiated use case requirements specified that it must be possible for the application to use the storage for reading, writing, removing and searching from the storage. As described in chapter 5.2, all of these requirements are fulfilled or supported by the component. In addition to the above, application must be able to initialize the storage contents from external data source. This

requirement is also covered using external initialization loaders as specified in chapter 5.2.5.

As a summary, all application initiated use case requirements were met.

6.1.3. Administrator use cases

Administration initiated use cases specified that it must be possible for the administrator to configure and monitor the distributed data storage component. Full integration with the service creation platform O&M services fulfils this requirement.

Additionally the use cases specified that it must be possible to add, remove, start-up and shutdown the nodes in the cluster without affecting the remaining cluster. All mentioned use cases are supported with integration to the service creation platform's Cluster subsystem, and by reacting immediately to the status changes provided by the Cluster subsystem. These use cases are also part of the tested functionality of the existing implementations presented in chapter 6.1.1.

As a summary, all administrator initiated use case requirements were met.

6.1.4. Failure use cases

Failure use cases specified that it must be possible for the storage to survive from uncontrolled application and node crashes. The design and implementation relies heavily on integration between Cluster subsystem, Distribution protocol and the storage component as described in chapter 5.2 and 6.1.3. The persistent storage and implemented recovery functionality ensures that the system can be configured in such a way that the data is not lost even in case of cluster-wide system failure.

As a summary, all failure use case requirements were met.

6.2. Functional Requirement Analysis

Following subchapters cover the analysis of functional requirements based on the implementation. The evaluation is based on the requirements given in chapter 4.2.

6.2.1. Storage requirements

Storage requirements specified that the storage must support storing of multiple java object types, and those java objects can be defined by the application. As specified in chapter 5.2 this requirements is covered by allowing Java objects to be stored to the storage, and by supporting multiple storage groups for different applications.

It was also required that the storage must provide high availability, transactional access, concurrency, local caching and persistency. The chapter 5.2 also provides details on how these requirements have been fulfilled. The requirements are also verified as part of the gateway message storage implementation described in chapter 6.1.1.

As a summary, all storage requirements were met.

6.2.2. Transaction distribution requirements

Transaction distribution requirements state that the storage must provide transparent and transaction distribution of storage data. Both transparency and transactional distribution is implemented inside the storage using the distribution protocol as described in chapter 5.2. As an extension of the requirements, in the current implementation the same API can also be used to run the storage either in distributed or local mode using the same API and exactly the same storage operations.

As a summary, all transaction distribution requirements were met.

6.2.3. Scalability requirements

Scalability requirements state that the storage shall be optimised for read access, and shall be capable of supporting alternative distribution models. The selected design uses Two-Phase Locking with Write-Locks-All distributed concurrency control. It is optimised for read access as described in chapter 3.3.3 and 3.3.5, having additional benefit of supporting serialised transaction order. The implementation also provides heap memory based local caching using LRU replacement policy, ensuring high performance reading from the storage.

The support for the alternative distribution models is implemented partially using the distribution protocol component in the service creation platform. The partial means that it is possible to provide alternative distribution policies using the distribution protocol component, but the core storage implementation does not yet know how to use them. Also the current implementation does not do proxy reading from other cluster nodes. It always expects to find the entries from its local storage.

As a summary, the basic scalability requirements were met. If alternative distributions models are taken into use, it requires some additional development on the distribution protocol and in some parts of the storage. This additional development is also described in chapter 7.1.

6.3. Non-Functional Requirement Analysis

Following subchapters cover the analysis of non-functional requirements based on the implementation. The evaluation is based on the requirements given in chapter 4.3.

Non-functional requirements state that storage shall be integrated to the platform and that it shall be extensible. Both requirements are covered by the full integration to the O&M services provided by the platform, and providing support for application specific interfaces for cache initialization loader and cache notifications, as well as application specific logics for indexing and searching.

The early tests regarding the performance indicate that the read performance is very fast (as expected) and is able to reach the required performance, but the distribution model limits the write performance.

As a summary, all non-functional requirements were met. There is already work on-going (as described in chapter 7.1) in order to resolve the scalability issues related with write performance. However, the design was optimised for read performance, and that has been reached as per requirement.

7. Conclusions

The distributed data storage implementation was done as part of the service creation platform development. The target for the service creation platform was to provide commonly used services implemented as reusable subsystems for the service applications running in operator networks. The target for the distributed data storage was to provide highly available data storage services to the service creation platform and the applications developed on top of the platform. Both targets have been met.

The design and the implementation is based on the requirements set for the component, and the decisions done during the work were influenced by the theory related to the transactions, concurrency control and data distribution. This background theory is presented in chapter 3.

The distributed data storage implementation has now been used in approximately five (5) different projects. Issues that have been found during the implementation projects have been resolved when encountered. So far the implementation has successfully met the targets set for the component although taken into use in use cases originally not targeted for. Following chapter describes some of the findings, and presents future work items to be done for improving the implementation.

7.1. Future work

The original requirement was to create a data storage, which provides high availability using transaction distribution and is optimised for read-only access. Lately, for example, during the gateway storage implementation project it has become clear that there is need to support scalability models that provide nearly linear scalability for writes also. There is currently an on-going activity to create a version of the distributed data storage, which provides better scalability for access models that consist of 50% writing (inserts, updated and deletes) and 50% reading.

Additional area of improvement is the support for large storage sizes. Currently all data is distributed between all nodes, which means that each node needs to be able to store all data that is stored into the storage. This can only scale up to certain limit, after which the data needs to be partitioned somehow in the

cluster. The early findings from the scalability improvement project seem to provide solution also to this problem.

Another known problem is in the implementation side. Currently all indexes are stored into the JVM heap memory, which of course makes access to the index faster, but it limits the size of the index and thus also the amount of entries that can be stored to the storage. The solution would be to implement new kind of indexing, which stores the index at least partially to local disk. One potential candidate for such an algorithm is the B-tree algorithm that is commonly used also in databases.

References

[1] Marcus, Evan; Stern, Hal: Blueprints for High Availability, John Wiley & Sons Inc., USA, 2000, 344 pages

[2] Service Delivery Platform, URL:http://en.wikipedia.org/wiki/Service_Delivery_Platform (9.2.2008)

[3] Piedad, Floyd; Hawkings, Michael: High Availability – Design, Techniques and Processes, Prentice Hall PTR, USA, 2001, 266 pages

[4] Jeffrey D. Ullman: Principles of Database and Knowledge – Base Systems, Volume I: Classical Database Systems, Computer Science Press, Inc., USA, 1988, 631 pages

[5] Jacobsen, Ivar; Booch, Grady; Rumbaugh, James: The Unified Software Development Process, Addison-Wesley, 1999, 463 pages

[6] ANSI/ISO/IEC International Standard (IS): Database Language SQL – Part 2: Foundation (SQL/Foundation), ISO/IEC 9075-2:1999, 1121 pages

[7] Gray, Jim; Reuter, Andreas: Transaction Processing: Concepts and Techniques, Morgan Kaufmann, 1993, 1070 pages

[8] Carey, Micheal; Livny, Miron: Distributed Concurrency Control Performance: A Study of Algorithms, Distribution and Replication, Proceedings of the 14th VLDB Conference, Los Angeles, California 1988

[9] K.M. Chandy; J. Misra; L.M. Haas: Distributed Deadlock Detection, ACM Transactions on Computer Systems, 1(2), May 1983, pages 143-156

[10] Coffman, E.G.; Elphick, M.J.; Shoshani, A. : System Deadlocks, ACM Computing Surveys, 3, 2, 1971, pages 67-78

[11] Bernstein, Philip A., Goodman, Nathan: Concurrency Control in Distributed Database Systems, Computing Surveys, Vol. 13, No. 2, June 1981

[12] Chang, Fay; Dean, Jeffrey; Ghemawat, Sanjay; Hsieh, Wilson C.; Wallach, Deborah A.; Burrows, Mike; Chandra, Tushar; Fikes, Andrew;

Gruber, Robert E.: Bigtable: A Distributed Storage System for Structured Data, http://labs.google.com/papers/bigtable-osdi06.pdf (31.3.2009)

[13] Sun Microsystems, Inc., Developer Resources for Java Technology, URL:http://java.sun.com/ (2.6.2009)

[14] Sun Microsystems, Inc., Sun Java Real-Time System, URL:http://java.sun.com/javase/technologies/realtime/index.jsp (8.6.2009)

[15] JCS – Java Caching System, URL:http://jakarta.apache.org/jcs/index.html (12.6.2009)

[16] Ehcache, URL:http://ehcache.sourceforge.net/ (13.6.2009)

[17] Jboss Community, Jboss Cache, URL:http://www.jboss.org/jbosscache/ (14.6.2009)

[18] Jboss Community, Infinispan, URL:http://www.jboss.org/infinispan/ (14.6.2009)

[19] Cache – Wikipedia, the free encyclopedia, URL: http://en.wikipedia.org/wiki/Cache (3.8.2009)

[20] Kent, Christopher: Cache Coherence in Distributed Systems, Western Research Laboratory, December 1987

[21] Peterson, James L.; Norman, Theodore A.: Buddy Systems, Department of Computer Sciences, The University of Texas At Austin, October 1974

[22] Sun Microsystems, Inc., JSLEE and the JAIN Initiative, URL: http://java.sun.com/products/jain/ (9.10.2009)

[23] Sun Microsystems, Inc., Java EE at a Glance, URL: http://java.sun.com/javaee/ (9.10.2009)

[24] Jouppi, Norman P.: Cache Write Policies and Performance, Digital Equipment Corporation, Western Research Lab, 20th ISCA, 1993, pages 191-201

[25] Bernstein, Philip A.; Goodman, Nathan: Timestamp-based Algorithms for Concurrency Control In Distributed Database Systems,

Proceedings of the 6th international conference on Very Large Data Bases – Volume 6, 1980, pages 285-300,

[26] Bernstein, Philip A.; Goodman, Nathan: Multiversion Concurrency Control – Theory and Algorithms, ACM Transactions on Database Systems, Vo. 8, No. 4, 1983, Pages 465-483

[27] Kung, H. T.; Robinson, John T.: On Optimistic Methods for Concurrency Control, ACM Transactions on Database Systems, Vol. 6, No. 2, June 1981, Pages 213-226

[28] Berenson, Hal; Bernstein, Phil; Gray, Jim; Melton, Jim; O'Neil, Elizabeth; O'Neil, Patrick: A Critique of ANSI SQL Isolation Levels, Proceedings of the 1996 ACM SIGMOD, Pages 173-182

[29] Open Source Initiative OSI, The BSD License, URL: http://www.opensource.org/licenses/bsd-license.php

[30] Free Software Foundation (FSF), GNU Lesser General Public License, URL:http://www.gnu.org/licenses/lgpl.html